Django Book 2.0 中文版

wizardforcel

Published
with GitBook



目錄

The Django Book 2.0--中文版

来源：docs.30c.org/djangobook2

整理：飞龙

第一章：介绍Django

本书所讲的是Django：一个可以使Web开发工作愉快并且高效的Web开发框架。 使用Django，使你能够以最小的代价构建和维护高质量的Web应用。

从好的方面来看，Web 开发激动人心且富于创造性；从另一面来看，它却是份繁琐而令人生厌的工作。 通过减少重复的代码，Django 使你能够专注于 Web 应用上有 趣的关键性的东西。 为了达到这个目标，Django 提供了通用Web开发模式的高度抽象，提供了频繁进行的编程作业的快速解决方法，以及为"如何解决问题"提供了清晰明了的约定。 同时，Django 尝试留下一些方法，来让你根据需要在framework之外来开发。

本书的目的是将你培养成Django专家。 主要侧重于两方面： 第一，我们深度解释 Django 到底做了哪些工作以及如何用她构建Web应用；第二，我们将会在适当的地方讨论更高级的概念，并解释如何 在自己的项目中高效的使用这些工具。 通过阅读此书，你将学会快速开发功能强大网站的技巧，并且你的代码将会十分 清晰，易于维护。 本书的代码清晰，易维护，通过学习，可以快速开发功能强大的网站。

框架是什麼?

Django 在新一代的 *Web*框架 中非常出色，为什么这么说呢？

为回答该问题，让我们考虑一下不使用框架设计 Python 网页应用程序的情形。 贯穿整本书，我们多次展示不使用框架实现网站基本功能的方法，让读者认识到框架开发的方便。（不使用框架，更多情况是没有合适的框架可用。 最重要的是，理解实现的来龙去脉会使你成为一个优秀的web开发者。）

使用Python开发Web，最简单，原始和直接的办法是使用CGI标准，在1998年这种方式很流行。 现在从应用角度解释它是如何工作： 首先做一个Python脚本，输出HTML代码，然后保存成.cgi扩展名的文件，通过浏览器访问此文件。 就是这样。

如下示例，用Python CGI脚本显示数据库中最新出版的10本书： 不用关心语法细节；仅仅感觉一下基本实现的方法：

```
#!/usr/bin/env python

import MySQLdb

print "Content-Type: text/html\n"
print "<html><head><title>Books</title></head>"
print "<body>"
print "<h1>Books</h1>"
print "<ul>"

connection = MySQLdb.connect(user='me', passwd='letmein', db='my_db')
cursor = connection.cursor()
cursor.execute("SELECT name FROM books ORDER BY pub_date DESC LIMIT 10")

for row in cursor.fetchall():
    print "<li>%s</li>" % row[0]

print "</ul>"
print "</body></html>"

connection.close()
```

首先，用户请求CGI，脚本代码打印Content-Type行，后面跟着换行。 再接下 来是一些HTML的起始标签，然后连接数据库并执行一些查询操作，获取最新的十本书。 在遍历这些书的同时，生成一个书名的HTML列表项。 最后，输出HTML的结束标签并且关闭数据库连接。

像这样的一次性的动态页面，从头写起的方法并非一定不好。 其中一点： 这些代码简单易懂，就算是一个初起步的 开发者都能读明白这16行的Python的代码，而且这些代码从头到尾做了什么都能了解得一清二楚。 不需要学习额外 的背景知识，没有额外的代码需要去了解。 同样，也易于部署这16行代码，只需要将它保存为一个 `latestbooks.cgi` 的 文件，上传到网络服务器上，通过浏览器访问即可。

尽管实现很简单，还是暴露了一些问题和不便的地方。 问你自己这几个问题：

- 应用中有多处需要连接数据库会怎样呢？ 每个独立的CGI脚本，不应该重复写数据库连接的代码。 比较实用的办法是写一个共享函数，可被多个代码调用。
- 一个开发人员 确实 需要去关注如何输出Content-Type以及完成所有操作后去关闭数据 库么？ 此类问题只会降低开发人员的工作效率，增加犯错误的几率。 那些初始化和释放 相关的工作应该交给一些通用的框架来完成。
- 如果这样的代码被重用到一个复合的环境中会发生什么？ 每个页面都分别对应独立的数据库和密码吗？
- 如果一个Web设计师，完全没有Python开发经验，但是又需要重新设计页面的话，又将发生什么呢？ 一个字符写错了，可能导致整个应用崩溃。 理想的情况是，页面显示的逻辑与从数据库中读取书本记录分隔开，这样 Web设计师的重新设计不会影响到之前的业务逻辑。

以上正是Web框架致力于解决的问题。 Web框架为应用程序提供了一套程序框架， 这样你可以专注于编写清晰、易维护的代码，而无需从头做起。 简单来说，这就是Django所能做的。

MVC 设计模式

让我们来研究一个简单的例子，通过该实例，你可以分辨出，通过Web框架来实现的功能与之前的方式有何不同。 下面就是通过使用Django来完成以上功能的例子： 首先，我们分成4个Python的文件，(`models.py` , `views.py` , `urls.py`) 和html模板文件 (`latest_books.html`)

```
# models.py (the database tables)

from django.db import models

class Book(models.Model):
    name = models.CharField(max_length=50)
    pub_date = models.DateField()

# views.py (the business logic)

from django.shortcuts import render_to_response
from models import Book

def latest_books(request):
    book_list = Book.objects.order_by('-pub_date')[:10]
    return render_to_response('latest_books.html', {'book_list': book_list})

# urls.py (the URL configuration)

from django.conf.urls.defaults import *
import views

urlpatterns = patterns('',
    (r'^latest/$', views.latest_books),
)

# latest_books.html (the template)

<html><head><title>Books</title></head>
<body>
<h1>Books</h1>
<ul>
{% for book in book_list %}
<li>{{ book.name }}</li>
{% endfor %}
</ul>
</body></html>
```

然后，不用关心语法细节；只要用心感觉整体的设计。 这里只关注分割后的几个文件：

- `models.py` 文件主要用一个 Python 类来描述数据表。 称为 模型*(model)* 。 运用这个类，你可以通过简单的 Python 的代码来创建、检索、更新、删除 数据库中的记录而无需写一条又一条的SQL语句。
- views.py文件包含了页面的业务逻辑。 `latest_books()` 函数叫做视图。
- `urls.py` 指出了什么样的 URL 调用什么的视图。 在这个例子中 `/latest/` URL 将会调用 `latest_books()` 这个函数。 换句话说，如果你的域名是example.com，任何人浏览网址http://example.com/latest/将会调用`latest_books()`这个函数。

- `latest_books.html` 是 html 模板，它描述了这个页面的设计是如何的。 使用带基本逻辑声明的模板语言，如 `{% for book in book_list %}`

结合起来，这些部分松散遵循的模式称为模型-视图-控制器(MVC)。 简单的说， MVC 是一种软件开发的方法，它把代码的定义和数据访问的方法（模型）与请求逻辑 （控制器）还有用户接口（视图）分开来。 我们将在第5章更深入地讨论MVC。

这种设计模式关键的优势在于各种组件都是 松散结合 的。这样，每个由 Django驱动 的Web应用都有着明确的目的，并且可独立更改而不影响到其它的部分。 比如，开发者 更改一个应用程序中的 URL 而不用影响到这个程序底层的实现。 设计师可以改变 HTML 页面 的样式而不用接触 Python 代码。 数据库管理员可以重新命名数据表并且只需更改一个地方，无需从一大堆文件中进行查找和替换。

本书中，每个组件都有它自己的一个章节。 比如，第三章涵盖了视图，第四章是模板， 而第五章是模型。

Django 历史

在我们讨论代码之前我们需要先了解一下 Django 的历史。 从上面我们注意到：我们将向你展示如何不使用捷径来完成工作，以便能更好的理解捷径的原理 同样，理解Django产生的背景， 历史有助于理解Django的实现方式。

如果你曾编写过网络应用程序。 那么你很有可能熟悉之前我们的 CGI 例子。

1. 从头开始编写网络应用程序。
2. 从头编写另一个网络应用程序。
3. 从第一步中总结（找出其中通用的代码），并运用在第二步中。
4. 重构代码使得能在第 2 个程序中使用第 1 个程序中的通用代码。
5. 重复 2-4 步骤若干次。
6. 意识到你发明了一个框架。

这正是为什么 Django 建立的原因！

Django 是从真实世界的应用中成长起来的，它是由 堪萨斯（Kansas）州 Lawrence 城中的一个 网络开发小组编写的。 它诞生于 2003 年秋天，那时 *Lawrence Journal-World* 报纸的程序员 Adrian Holovaty 和 Simon Willison 开始用 Python 来编写程序。

当时他们的 World Online 小组制作并维护当地的几个新闻站点, 并在以新闻界特有的快节奏开发环境中逐渐发展。 这些站点包括有 LJWorld.com、Lawrence.com 和 KUsports.com， 记者（或管理层） 要求增加的特征或整个程序都能在计划时间内快速的被建立，这些时间通常

只有几天 或几个小时。 因此，Adrian 和 Simon 开发了一种节省时间的网络程序开发框架，这是在截止时间前能完成程序的唯一途径。

2005 年的夏天，当这个框架开发完成时，它已经用来制作了很多个 World Online 的站点。当时 World Online 小组中的 Jacob Kaplan-Moss 决定把这个框架发布为一个开源软件。

从今往后数年，Django是一个有着数以万计的用户和贡献者，在世界广泛传播的完善开源项目。 原来的World Online的两个开发者（Adrian and Jacob）仍然掌握着Django，但是其发展方向受社区团队的影响更大。

这些历史都是相关联的，因为她们帮助解释了很重要的两点。 第一，Django最可爱的地方。Django诞生于新闻网站的环境中，因此它提供很多了特性（如第6章会说到的管理后台），非常适合内容类的网站，如Amazon.com, craigslist.org和washingtonpost.com，这些网站提供动态的，数据库驱动的信息。 （不要看到这就感到沮丧，尽管Django擅长于动态内容管理系统， 但并不表示Django主要的目的就是用来创建动态内容的网站。 某些方面 特别高效 与其他方面 不高效 是有区别的, Django在其他方面也同样高效。）

第二，Django的起源造就了它的开源社区的文化。 因为Django来自于真实世界中的代码，而不是 来自于一个科研项目或者商业产品，她主要集中力量来解决Web开发中遇到的问题，同样 也是Django的开发者经常遇到的问题。 这样，Django每天在现有的基础上进步。 框架的开发者对于让开发人员节省时间，编写更加容易维护的程序，同时保证程序运行的效率具有极大的兴趣。 无他，开发者动力来源于自己的目标：节省时间，快乐工作。 （坦率地讲，他们使用了自己公司的产品。）

如何阅读本书

在编写本书时，我们努力尝试在可读性和参考性间做一个平衡，当然本书会偏向于可 读性。本书的目标，之前也提过，是要将你培养成一名Django专家，我们相信，最好 的方式就是提供文章和充足的实例，而不是一堆详尽却乏味的关于Django特色的手册。 （曾经有人说过，如果仅仅教字母表是无法教会别人说话的。

按照这种思路，我们推荐按顺序阅读第 1-12 章。 这些章节构成了如何使用 Django 的基础；读过之后，你就可以搭建由 Django 支撑的网站了。 1-7章是核心课程，8-11章讲述Django的高级应用，12章讲述部署相关的知识。 剩下的13-20章，讲述Django特有的特点，可以任意顺序阅读。

附录部分用作参考资料。 要回忆语法或查阅 Django 某部分的功能概要时，你偶尔可能会回来翻翻这些资料以及 http://www.djangoproject.com/ 上的免费文档。

所需编程知识

本书读者需要理解基本的面向过程和面向对象编程： 流程控制（ `if` ， `while` 和 `for` ），数据结构（列表，哈希表/字典），变量，类和对象。

Web开发经验，正如你所想的，也是非常有帮助的，但是对于阅读本书，并不是必须的。 通过本书，我们尽量给缺乏经验的开发人员提供在Web开发中最好的实践。

Python所需知识

本质上来说， Django 只不过是用 Python 编写的一组类库。 用 Django 开发站点就是使用这些类库编写 Python 代码。 因此，学习 Django 的关键就是学习如何进行 Python 编程并理解 Django 类库的运作方式。

如果你有Python开发经验，在学习过程中应该不会有任何问题。 基本上,Django的代码并 没有使用一些黑色魔法（例如代码中的花哨技巧，某个实现解释或者理解起来十分困难）。 对你来说，学习Django就是学习她的命名规则和API。

如果你没有使用 Python 编程的经验，你一定会学到很多东西。 它是非常易学易用的。 虽然这本书没有包括一个完整的 Python 教程， 但也算是一个恰当的介绍了 Python特征和 功能的集锦。 当然，我们推荐你读一下官方的 Python 教程，它可 以从 http://docs.python.org/tut/ 在线获得。 另外我们也推荐 Mark Pilgrims的 书 *Dive Into Python* （ http://www.diveintopython.org/ ）

Django版本支持

此书内容对Django 1.1兼容。

Django的开发者保证主要版本号向后兼容。 这意味着，你用Django 1.1写的应用，可以用于1.2，1.3，1.9等所有以1开头的版本

如果Django到了2.0，你的应用可能不再兼容，需要重写，但是，2.0是很遥远的事情。 对此，可以参考一下1.0的开发周期，整整3年的时间。 （这与Python语言的兼容策略非常像：在python 2.0下写的代码可以在python 2.6下运行，但不一定能在python3.0下运行

所以，此书覆盖1.1版本，可以使用很长时间。

获取帮助

Django的最大的益处是,有一群乐于助人的人在Django社区上。 你可以毫无约束的提各种 问题在上面,如:django的安装,app 设计,db 设计,发布。

- Django邮件列表是很多Django用户提出问题、回答问题的地方。 可以通过 http://www.djangoproject.com/r/django-users 来免费注册。
- 如果Django用户遇到棘手的问题,希望得到及时地回复，可以使用Django IRC channel。在Freenode IRC network加入#django

下一章

在 下一章，我们将开始使用Django，内容将包括安装和初始化配置。

第二章：入门

由于现代Web开发环境由多个部件组成，安装Django需要几个步骤。 这一章，我们将演示如何安装框架以及一些依赖关系。

因为Django就是纯Python代码，它可以运行在任何Python可以运行的环境，甚至是手机上！但是这章只提及Django安装的通用脚本。 我们假设你把它安装在桌面/笔记本电脑或服务器。

往后，在第12章，我们将讨论如何部署Django到一个生产站点。

Python 安装

Django本身是纯Python编写的，所以安装框架的第一步是确保你已经安装了Python。

Python版本

核心Django框架可以工作在2.3至2.6（包括2.3和2.6）之间的任何Python版本。 Django的可选GIS（地理信息系统）支持需要Python 2.4到2.6。

如果你不确定要安装Python的什么版本，并且你完全拿不定主意的话,那就选2.x系列的最新版本吧。 版本2.6。 虽然Django在2.3至2.6版之间的任意Python版本下都一样运行得很好，但是新版本的Python提供了一些你可能比较想应用在你的程序里的，更加丰富和额外的语言特性。 另外，某些你可能要用到的Django第三方插件会要求比Python 2.3更新的版本，所以使用比较新的Python版本会让你有更多选择。

Django和 Python 3.0

在写作本书的时候，Python3.0已经发布，但Django暂时还不支持。 Python3.0这个语言本身引入了大量不向后兼容的改变，因此，我们预期大多数主要的Python库和框架将花几年才能衔接，包括Django。

如果你是个Python新手并且正迷茫于到底是学习Python 2.x还是Python 3.x的话，我们建议你选择Python 2.x。

安装

如果使用的是 Linux 或 Mac OS X ，系统可能已经预装了 Python 。在命令提示符下 (或 OS X 的终端中) 输入 `python` ，如果看到如下信息，说明 Python 已经装好了： 在命令行窗口中输入 `python` （或是在OS X的程序/工具/终端中）。 如果你看到这样的信息,说明 python 已经安装好了.

```
Python 2.4.1 (#2, Mar 31 2005, 00:05:10)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1666)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

否则, 你需要下载并安装Python. 它既快速又方便，而详细说明可参考http://www.python.org/download/

安装 Django

任何时候，都有两个不同版本的Django供您选择。 最新的官方发行版和有风险的主干版本。 安装的版本取决于您的优先选择。 你需要一个稳定的通过测试的Django，或是你想要包括最新功能的版本，也许你可对Django本身作贡献，而把稳定作为代价？

我们推荐选定一个正式发布版本，但重要的是了解到主干开发版本的存在，因为在文档和社区成员中你会发现它被提到。

安装官方发布版

官方发布的版本带有一个版本号，例如1.0.3或1.1，而最新版本总是可以在http://www.djangoproject.com/download/找到。

如果您在用Linux系统，其中包括Django的包，使用默认的版本是个好主意。 这样，你将会通过系统的包管理得到安全的升级。

如果你的系统没有自带Django，你可以自己下载然后安装框架。 首先，下载名字类似于 `Django-1.0.2-final.tar.gz` 压缩文件。 （下载到哪里无所谓，安装程序会把Django文件放到正确的地方。）解压缩之后运行 `setup.py install` ，像操作大多数Python库一样。

以下是如何在Unix系统上安装的方法：

1. `tar xzvf Django-*.tar.gz` 。
2. `cd Django-*` 。
3. `sudo python setup.py install` 。

Windows系统上，推荐使用7-Zip(http://www.djangoproject.com/r/7zip/)来解压缩`.tar.gz`文件。 解压缩完成后，以管理员权限启动一个DOS Shell（命令提示符），然后在名字以 `Django-` 开始的目录里执行如下命令：

```
python setup.py install
```

如果你很好奇： Django将被安装到你的Python安装目录 `的site-package` 目录（Python从该目录寻找第三方库）。 通常情况下，这个目录在 `/usr/lib/python2.4/site-packages` 。

安装Trunk版本

最新最好的django的开发版本称为trunk,可以从django的subversion处获得。 如果你想尝鲜，或者想为django贡献代码，那么你应当安装这个版本。

Subversion 是一种与 CVS 类似的免费开源版本控制系统，Django 开发团队使用它管理 Django 代码库的更新。 你可以使用 Subversion 客户端获取最新的 Django 源代码，并可任何时候使用 *local checkout* 更新本地 Django 代码的版本，以获取 Django 开发者所做的最近更新和改进。

请记住，即使是使用trunk版本，也是有保障的。 因为很多django的开发者在正式网站上就是用的trunk版本，他们会保证trunk版本的稳定性。

遵循以下步骤以获取最新的 Django 主流代码：

> 确保安装了 Subversion 客户端。 可以从 http://subversion.tigris.org/ 免费下载该软件，并从 http://svnbook.red-bean.com/ 获取出色的文档。
>
> （如果你在使用Mac OS X 10.5或者更新的版本，你很走运，Subversion应该就可以安装Django。 你可以在终端上输入 `svn --version` 来验证。
>
> 使用 `svn co http://code.djangoproject.com/svn/django/trunk djtrunk` 命令查看主体代码。
>
> 找到你的python的site-packages目录。 一般为/usr/lib/python2.4/site-packages，如果你不确定，可以输入如下命令：

```
python -c 'import sys, pprint; pprint.pprint(sys.path)'
```

> 上面的结果会包含site-packages的目录
>
> 在site-packages目录下，创建一个文件
>
> > django.pth，编辑这个文件，包含djtrunk目录的全路径 利润，此文件包含如下行：

```
/home/me/code/djtrunk
```

1. 将 `djtrunk/django/bin` 加入系统变量 PATH 中。该目录中包含一些像 `django-admin.py` 之类的管理工具。 此目录包含管理工具，例如：django-admin.py

提示：

如果之前没有接触过 `.pth` 文件，你可以从 http://www.djangoproject.com/r/python/site-module/ 中获取更多相关知识。

从 Subversion 完成下载并执行了前述步骤后，就没有必要再执行 `python setup.py install` 了，你刚才已经手动完成了安装！

由于 Django 主干代码的更新经常包括 bug 修正和特性添加，如果真的着迷的话，你可能每隔一小段时间就想更新一次。 在 `djtrunk` 目录下运行 `svn update` 命令即可进行更新。 当你使用这个命令时，Subversion 会联络 http://code.djangoproject.com ，判断代码是否有更新，然后把上次更新以来的所有变动应用到本地代码。 就这么简单。

最后，如果你使用trunk，你要知道使用的是哪个trunk版本。 如果你去社区寻求帮助，或是为Django框架提供改进，知道你使用的版本号是非常重要的。 因此，当你到社区去求助，或者为 django 提供改进意见的时候，请时刻记住说明你正在使用的 django 的版本号。 如何知道你正在使用的 django 的版本号呢？进入 `djtrunk` 目录，然后键入 svn info ，在输出信息中查看 Revision: (版本:) 后跟的数字。 Django在每次更新后，版本号都是递增的，无论是修复Bug、增加特性、改进文档或者是其他。 在一些Django社区中，版本号甚至成为了一种荣誉的象征，我从[写上非常低的版本号]开始就已经使用Djano了。

测试Django安装

让我们花点时间去测试 Django 是否安装成功，并工作良好。同时也可以了解到一些明确的安装后的反馈信息。 在Shell中，更换到另外一个目录（不是包含Django的目录），然后输入 `python` 来打开Python的交互解释器。如果安装成功，你应该可以导入 `django` 模块了：

```
>>> import django
>>> django.VERSION
(1, 1, 0, final', 1)
```

交互解释器示例

Python 交互解释器是命令行窗口的程序，通过它可以交互式地编写 Python 程序。 要启动它只需运行 `python` 命令。

我们在交互解释器中演示Python示例将贯穿整本书。 你可以用三个大于号 (`>>>`)来分辨出示例，三个大于号就表示交互提示符。 如果你要从本书中拷贝示例，请不要拷贝提示符。

在交互式解释器中，多行声明用三个点 (`...`) 来填补。 例如：

```
>>> print """This is a
... string that spans
... three lines."""
This is a
string that spans
three lines.
>>> def my_function(value):
...     print value
>>> my_function('hello')
hello
```

这三个在新行开始插入的点，是Python Shell自行加入的，不属于我们的输入。 但是包含它们是为了追求解释器的正确输出。 如果你拷贝我们的示例去运行，千万别拷贝这些点。

安装数据库

这会儿，你可以使用django写web应用了，因为django只要求python正确安装后就可以跑起来了。 不过，当你想开发一个数据库驱动的web站点时，你应当需要配置一个数据库服务器。

如果你只想玩一下，可以不配置数据库，直接跳到 开始一个project 部分去，不过你要注意本书的例子都是假设你配置好了一个正常工作的数据库。

Django支持四种数据库：

- PostgreSQL (http://www.postgresql.org/)
- SQLite 3 (http://www.sqlite.org/)
- MySQL (http://www.mysql.com/)
- Oracle (http://www.oracle.com/)

大部分情况下，这四种数据库都会和Django框架很好的工作。 （一个值得注意的例外是Django的可选GIS支持，它为PostgreSQL提供了强大的功能。）如果你不准备使用一些老旧系统，而且可以自由的选择数据库后端，我们推荐你使用PostgreSQL，它在成本、特性、速度和稳定性方面都做的比较平衡。

设置数据库只需要两步：

- 首先，你需要安装和配置数据库服务器本身。 这个过程超出了本书的内容，不过这四种数据库后端在它的网站上都有丰富的文档说明。 如果你使用的是共享主机，可能它们已经为你设置好了。
- 其次，你需要为你的服务器后端安装必要的Python库。 这是一些允许Python连接数据库的第三方代码。 我们会在之后的章节简要介绍，对于某一种数据库来说，它单独需要安装的东西。

如果你只是玩一下，不想安装数据库服务，那么可以考虑使用SQLite。 如果你用python2.5或更高版本的话，SQLite是唯一一个被支持的且不需要以上安装步骤的数据库。 它仅对你的文件系统中的单一文件读写数据，并且Python2.5和以后版本内建了对它的支持。

在Windows上，取得数据库驱动程序可能会令人沮丧。 如果你急着用它，我们建议你使用python2.5。

在 Django 中使用 PostgreSQL

使用 PostgreSQL 的话，你需要从 http://www.djangoproject.com/r/python-pgsql/ 下载 `psycopg` 这个开发包。 我们建议使用 `psycopg2` ，因为它是新的，开发比较积极，且更容易安装。 留意你所用的是 版本 1 还是 2，稍后你会需要这项信息。

如果在 Windows 平台上使用 PostgreSQL，可以从 http://www.djangoproject.com/r/python-pgsql/windows/ 获取预编译的 `psycopg` 开发包的二进制文件。

如果你在用Linux，检查你的发行版的软件包管理系统是否提供了一套叫做python-psycopg2，psycopg2-python，python-postgresql这类名字的包。

在 Django 中使用 SQLite 3

如果你正在使用Python 2.5版本或者更高，那么你很幸运 ： 不要求安装特定的数据库，因为Python支持和SQLite进行通信。 向前跳到下一节。

如果你用的是Python2.4或更早的版本，你需要 SQLite 3而不是版本2，这个可从 http://www.djangoproject.com/r/sqlite/`pysqlite`http://www.djangoproject.com/r/python-sqlite/ 确认一下你的 `pysqlite` 版本是2.0.3或者更高。

在 Windows 平台上，可以跳过单独的 SQLite 二进制包安装工作，因为它们已被静态链接到 `pysqlite` 二进制开发包中。

如果你在用Linux，检查你的发行版的软件包管理系统是否提供了一套叫做python-sqlite3，sqlite-python，pysqlite这类名字的包。

在 Django 中使用 MySQL

django要求MySQL4.0或更高的版本。 3.X 版本不支持嵌套子查询和一些其它相当标准的SQL语句。

你还需要从 http://www.djangoproject.com/r/python-mysql/ 下载安装 `MySQLdb` 。

如果你正在使用Linux，检查下你系统的包管理器是否提供了叫做python-mysql,python-mysqldb,myspl-python或者相似的包。

在Django中使用Oracle数据库

django需要Oracle9i或更高版本。

如果你用Oracle，你需要安装 `cx_Oracle` 库,可以从http://cx-oracle.sourceforge.net/获得。 要用4.3.1或更高版本，但要避开5.0，这是因为这个版本的驱动有bug。

使用无数据库支持的 Django

正如之前提及过的，Django 并不是非得要数据库才可以运行。 如果只用它提供一些不涉及数据库的动态页面服务，也同样可以完美运行。

尽管如此，还是要记住：

> Django 所捆绑的一些附加工具 一定 需要数据库，因此如果选择不使用数据库，你将不能使用那些功能。 (我们将在本书中自始至终强调这些功能)

开始一个项目

一但你安装好了python，django和（可选的）数据库及相关库，你就可以通过创建一个*project*，迈出开发django应用的第一步。

项目 是 Django 实例的一系列设置的集合，它包括数据库配置、Django 特定选项以及应用程序的特定设置。

如果第一次使用 Django，必须进行一些初始化设置工作。 新建一个工作目录，例如 `/home/username/djcode/` ，然后进入该目录。

这个目录应该放哪儿？

有过 PHP 编程背景的话，你可能习惯于将代码都放在 Web 服务器的文档根目录 (例如 `/var/www` 这样的地方)。 而在 Django 中，把任何Python代码和web server的文档根(root)放在一起并不是一个好主意。因为这样做有使人能通过网路看到你原代码的风险. 那就太糟了。

把代码放置在文档根目录 之外 的某些目录中。

转到你创建的目录，运行命令 `django-admin.py startproject mysite` 。这样会在你的当前目录下创建一个目录。 `mysite`

注意

如果用的是 `setup.py` 工具安装的 Django ， `django-admin.py` 应该已被加入了系统路径中。

如果你使用一个trunk版本，你会在 `djtrunk/django/bin` 下发现 `django-admin.py` 。你将来会常用到 `django-admin.py` ，考虑把它加到你的系统路径中去比较好。 在Unix中, 你也可以用来自 `/usr/local/bin` 的符号连接, 用一个命令, 诸如 `sudo ln -s /path/to/django/bin/django-admin.py /usr/local/bin/django-admin.py` . 在Windows中, 你需要修改你的 `PATH` 环境变量.

如果你的django是从linux发行版中安装的，那么，常会被 `django-admin.py` 替代。 `django-admin`

如果在运行时，你看到权限拒绝的提示，你应当修改这个文件的权限。`django-admin.py startproject` 为此, 键入 `cd /usr/local/bin` 转到 `django-admin.py` 所在的目录，运行命令 `chmod +x django-admin.py`

`startproject` 命令创建一个目录，包含4个文件：

```
mysite/
    __init__.py
    manage.py
    settings.py
    urls.py
```

文件如下：

- `__init__.py` ：让 Python 把该目录当成一个开发包 (即一组模块)所需的文件。 这是一个空文件，一般你不需要修改它。
- `manage.py` ：一种命令行工具，允许你以多种方式与该 Django 项目进行交互。 键入 `python manage.py help` ，看一下它能做什么。 你应当不需要编辑这个文件；在这个目录下生成它纯是为了方便。
- `settings.py` ：该 Django 项目的设置或配置。 查看并理解这个文件中可用的设置类型及其默认值。
- `urls.py` ：Django项目的URL设置。 可视其为你的django网站的目录。 目前，它是空的。

尽管这些的文件很小，但这些文件已经构成了一个可运行的Django应用。

运行开发服务器

为了安装后更多的体验，让我们运行一下django开发服务器看看我们的准系统。

django开发服务是可用在开发期间的，一个内建的，轻量的web服务。 我们提供这个服务器是为了让你快速开发站点，也就是说在准备发布产品之前，无需进行产品级 Web 服务器（比如 Apache）的配置工作。 开发服务器监测你的代码并自动加载它，这样你会很容易修改代码而不用重启动服务。

如果你还没启动服务器的话，请切换到你的项目目录里 (`cd mysite`)，运行下面的命令：

```
python manage.py runserver
```

你会看到些像这样的

```
Validating models...
0 errors found.

Django version 1.0, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

这将会在端口8000启动一个本地服务器, 并且只能从你的这台电脑连接和访问。 既然服务器已经运行起来了，现在用网页浏览器访问 http://127.0.0.1:8000/ 。 你应该可以看到一个令人赏心悦目的淡蓝色Django欢迎页面。 它开始工作了。

在进一步学习之前， 一个重要的，关于开发网络服务器的提示很值得一说。 虽然 django 自带的这个 web 服务器对于开发很方便，但是，千万不要在正式的应用布署环境中使用它。 在同一时间，该服务器只能可靠地处理一次单个请求，并且没有进行任何类型的安全审计。 发布站点前，请参阅第 20 章了解如何部署 Django 。

更改这个 Development Server 的主机地址或端口

默认情况下， `runserver` 命令在 8000 端口启动开发服务器，且仅监听本地连接。 要想要更改服务器端口的话，可将端口作为命令行参数传入：

```
python manage.py runserver 8080
```

通过指定一个 IP 地址，你可以告诉服务器–允许非本地连接访问。 如果你想和其他开发人员共享同一开发站点的话，该功能特别有用。 `0.0.0.0` 这个 IP 地址，告诉服务器去侦听任意的网络接口。

```
python manage.py runserver 0.0.0.0:8000
```

完成这些设置后，你本地网络中的其它计算机就可以在浏览器中访问你的 IP 地址了。比如：http://192.168.1.103:8000/ . (注意，你将需要校阅一下你的网络配置来决定你在本地网络中的IP 地址) Unix用户可以在命令提示符中输入ifconfig来获取以上信息。 使用Windows的用户，请尝试使用 ipconfig 命令。

接下来做什么?

好了，你已经安装好所需的一切， 并且开发服务器也运行起来了，你已经准备好继续 学习基础知识–用Django伺候网頁 这一章的内容了。

第三章：视图和URL配置

前一章中，我们解释了如何建立一个 Django 项目并启动 Django 开发服务器。 在这一章，你将会学到用Django创建动态网页的基本知识。

你的第一个基于Django的页面： Hello World

正如我们的第一个目标，创建一个网页，用来输出这个著名的示例信息：

Hello world.

如果你曾经发布过Hello world页面，但是没有使用网页框架，只是简单的在 `hello.html` 文本文件中输入Hello World，然后上传到任意的一个网页服务器上。 注意，在这个过程中，你已经说明了两个关于这个网页的关键信息： 它包括（字符串 `"Hello world"` ）和它的URL(`http://www.example.com/hello.html` ，如果你把文件放在子目录，也可能是 `http://www.example.com/files/hello.html`)。

使用Django，你会用不同的方法来说明这两件事 页面的内容是靠*view function*（视图函数）来产生，URL定义在 *URLconf* 中。首先，我们先写一个Hello World视图函数。

第一份视图：

在上一章使用 `django-admin.py startproject` 制作的 `mysite` 文件夹中，创建一个叫做 `views.py` 的空文件。这个Python模块将包含这一章的视图。 请留意，Django对于view.py的文件命名没有特别的要求，它不在乎这个文件叫什么。但是根据约定，把它命名成view.py是个好主意，这样有利于其他开发者读懂你的代码，正如你很容易的往下读懂本文。

我们的Hello world视图非常简单。 这些是完整的函数和导入声明，你需要输入到 `views.py` 文件：

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

我们逐行逐句地分析一遍这段代码：

> 首先，我们从 `django.http` 模块导入（import） `HttpResponse` 类。参阅附录 H 了解更多关于 `HttpRequest` 和 `HttpResponse` 的细节。 我们需要导入这些类，因为我们会在后面用到。
>
> 接下来，我们定义一个叫做`hello` 的视图函数。
>
> 每个视图函数至少要有一个参数，通常被叫作`request` 。 这是一个触发这个视图、包含当前Web请求信息的对象，是类`django.http.HttpRequest`的一个实例。在这个示例中，我们虽然不用`request`做任何事情，然而它仍必须是这个视图的第一个参数。
>
> 注意视图函数的名称并不重要；并不一定非得以某种特定的方式命名才能让 Django 识别它。 在这里我们把它命名为：hello，是因为这个名称清晰的显示了视图的用意。同样地，你可以用诸如：hello_wonderful_beautiful_world，这样难看的短句来给它命名。 在下一小节（Your First URLconf），将告诉你Django是如何找到这个函数的。
>
> 这个函数只有简单的一行代码： 它仅仅返回一个HttpResponse对象，这个对象包含了文本“Hello world”。

这里主要讲的是： 一个视图就是Python的一个函数。这个函数第一个参数的类型是HttpRequest；它返回一个HttpResponse实例。为了使一个Python的函数成为一个Django可识别的视图，它必须满足这两个条件。 （也有例外，但是我们稍后才会接触到。

你的第一个URLconf

现在，如果你再运行：python manage.py runserver，你还将看到Django的欢迎页面，而看不到我们刚才写的Hello world显示页面。 那是因为我们的mysite项目还对hello视图一无所知。我们需要通过一个详细描述的URL来显式的告诉它并且激活这个视图。 （继续我们刚才类似发布静态HTML文件的例子。现在我们已经创建了HTML文件，但还没有把它上传至服务器的目录。）为了绑定视图函数和URL，我们使用URLconf。

URLconf 就像是 Django 所支撑网站的目录。 它的本质是 URL 模式以及要为该 URL 模式调用的视图函数之间的映射表。 你就是以这种方式告诉 Django，对于这个 URL 调用这段代码，对于那个 URL 调用那段代码。 例如，当用户访问/foo/时，调用视图函数foo_view()，这个视图函数存在于Python模块文件view.py中。

前一章中执行 `django-admin.py startproject` 时，该脚本会自动为你建了一份 URLconf（即 `urls.py` 文件）。 默认的urls.py会像下面这个样子：

```
from django.conf.urls.defaults import *

# Uncomment the next two lines to enable the admin:
# from django.contrib import admin
# admin.autodiscover()

urlpatterns = patterns('',
    # Example:
    # (r'^mysite/', include('mysite.foo.urls')),

    # Uncomment the admin/doc line below and add 'django.contrib.admindocs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # (r'^admin/', include(admin.site.urls)),
)
```

默认的URLconf包含了一些被注释起来的Django中常用的功能，仅仅只需去掉这些注释就可以开启这些功能. 下面是URLconf中忽略被注释的行后的实际内容

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
)
```

让我们逐行解释一下代码：

- 第一行导入django.conf.urls.defaults下的所有模块，它们是Django URLconf的基本构造。 这包含了一个patterns函数。
- 第二行调用 `patterns()` 函数并将返回结果保存到 `urlpatterns` 变量。patterns函数当前只有一个参数—一个空的字符串。 （这个字符串可以被用来表示一个视图函数的通用前缀。具体我们将在第八章里面介绍。）

当前应该注意是 `urlpatterns` 变量， Django 期望能从 `ROOT_URLCONF` 模块中找到它。 该变量定义了 URL 以及用于处理这些 URL 的代码之间的映射关系。 默认情况下，URLconf 所有内容都被注释起来了——Django 应用程序还是白版一块。 （注：那是上一节中Django怎么知道显示欢迎页面的原因。 如果 URLconf 为空，Django 会认定你才创建好新项目，因此也就显示那种信息。

如果想在URLconf中加入URL和view，只需增加映射URL模式和view功能的Python tuple即可. 这里演示如何添加view中hello功能.

```
from django.conf.urls.defaults import *
from mysite.views import hello

urlpatterns = patterns('',
    ('^hello/$', hello),
)
```

请留意：为了简洁，我们移除了注释代码。 如果你喜欢的话，你可以保留那些行。）

我们做了两处修改。

- 首先，我们从模块 (在 Python 的 import 语法中， `mysite/views.py` 转译为 `mysite.views`) 中引入了 `hello` 视图。 （这假设mysite/views.py在你的Python搜索路径上。关于搜索路径的解释，请参照下文。）
- 接下来，我们为urlpatterns加上一行： ('^hello/$', hello), 这行被称作URLpattern，它是一个Python的元组。元组中第一个元素是模式匹配字符串（正则表达式）；第二个元素是那个模式将使用的视图函数。

简单来说，我们只是告诉 Django，所有指向 URL `/hello/` 的请求都应由 `hello` 这个视图函数来处理。

Python 搜索路径

Python 搜索路径 就是使用 `import` 语句时，Python 所查找的系统目录清单。

举例来说，假定你将 Python 路径设置为 `['','/usr/lib/python2.4/site-packages','/home/username/djcode/']` 。如果执行代码 `from foo import bar` ，Python 将会首先在当前目录查找 `foo.py` 模块(Python 路径第一项的空字符串表示当前目录)。 如果文件不存在，Python将查找 `/usr/lib/python2.4/site-packages/foo.py` 文件。

如果你想看Python搜索路径的值，运行Python交互解释器，然后输入：

```
>>> import sys
>>> print sys.path
```

通常，你不必关心 Python 搜索路径的设置。 Python 和 Django 会在后台自动帮你处理好。

讨论一下URLpattern的语法是值得的，因为它不是显而易见的。 虽然我们想匹配地址/hello/，但是模式看上去与这有点差别。 这就是为什么：

> Django在检查URL模式前，移除每一个申请的URL开头的斜杠(/)。 这意味着我们为/hello/写URL模式不用包含斜杠(/)。（刚开始，这样可能看起来不直观，但这样的要求简化了许多工作，如URL模式内嵌，我们将在第八章谈及。）
>
> 模式包含了一个尖号(^)和一个美元符号($)。这些都是正则表达式符号，并且有特定的含义： 上箭头要求表达式对字符串的头部进行匹配，美元符号则要求表达式对字符串的尾部进行匹配。
>
> 最好还是用范例来说明一下这个概念。 如果我们用尾部不是$的模式'^hello/'，那么任何以/hello/开头的URL将会匹配，例如：/hello/foo 和/hello/bar，而不仅仅是/hello/。类似地，如果我们忽略了尖号(^)，即'hello/$'，那么任何以hello/结尾的URL将会匹配，例如：/foo/bar/hello/。如果我们简单使用hello/，即没有^开头和$结尾，那么任何包含hello/的URL将会匹配，如：/foo/hello/bar。因此，我们使用这两个符号以确保只有/hello/匹配，不多也不少。
>
> 你大多数的URL模式会以^开始、以$结束，但是拥有复杂匹配的灵活性会更好。
>
> 你可能会问：如果有人申请访问/hello（尾部没有斜杠/）会怎样。 因为我们的URL模式要求尾部有一个斜杠(/)，那个申请URL将不匹配。 然而，默认地，任何不匹配或尾部没有斜杠(/)的申请URL，将被重定向至尾部包含斜杠的相同字眼的URL。 （这是受配置文件setting中APPEND_SLASH项控制的，参见附件D。）
>
> 如果你是喜欢所有URL都以'/'结尾的人（Django开发者的偏爱），那么你只需要在每个URL后添加斜杠，并且设置"APPEND_SLASH"为"True". 如果不喜欢URL以斜杠结尾或者根据每个URL来决定，那么需要设置"APPEND_SLASH"为"False",并且根据你自己的意愿来添加结尾斜杠/在URL模式后.

另外需要注意的是，我们把hello视图函数作为一个对象传递，而不是调用它。 这是 Python (及其它动态语言的) 的一个重要特性： 函数是一级对象（first-class objects）， 也就是说你可以像传递其它变量一样传递它们。 很酷吧？

启动Django开发服务器来测试修改好的 URLconf, 运行命令行 `python manage.py runserver` 。 (如果你让它一直运行也可以，开发服务器会自动监测代码改动并自动重新载入，所以不需要手工重启） 开发服务器的地址是 http://127.0.0.1:8000/ ，打开你的浏览器访问 http://127.0.0.1:8000/hello/ 。 你就可以看到输出结果了。 开发服务器将自动检测Python代码的更改来做必要的重新加载， 所以你不需要重启Server在代码更改之后。服务器运行地址 `http://127.0.0.1:8000/` ，所以打开浏览器直接输入 `http://127.0.0.1:8000/hello/` ，你将看到由你的Django视图输出的Hello world。

万岁！ 你已经创建了第一个Django的web页面。

正则表达式

正则表达式 (或 *regexes*) 是通用的文本模式匹配的方法。 Django URLconfs 允许你 使用任意的正则表达式来做强有力的URL映射，不过通常你实际上可能只需要使用很少的一 部分功能。 这里是一些基本的语法。

符号	匹配
`.` (dot)	任意单一字符
`\d`	任意一位数字
`[A-Z]`	`A` 到 `Z`中任意一个字符（大写）
`[a-z]`	`a` 到 `z`中任意一个字符（小写）
`[A-Za-z]`	`a` 到 `z`中任意一个字符（不区分大小写）
`+`	匹配一个或更多 (例如, `\d+` 匹配一个或 多个数字字符)
`[^/]+`	一个或多个不为‘/’的字符
`*`	零个或一个之前的表达式（例如：`\d?` 匹配零个或一个数字）
`*`	匹配0个或更多 (例如, `\d*` 匹配0个 或更多数字字符)
`{1,3}`	介于一个和三个（包含）之前的表达式（例如，`\d{1,3}`匹配一个或两个或三个数字）

有关正则表达式的更多内容，请访问 http://www.djangoproject.com/r/python/re-module/.

关于“404错误”的快速参考

目前，我们的URLconf只定义了一个单独的URL模式： 处理URL `/hello/` 。 当请求其他URL会怎么样呢？

让我们试试看，运行Django开发服务器并访问类似 `http://127.0.0.1:8000/goodbye/` 或者 `http://127.0.0.1:8000/hello/subdirectory/` ，甚至 `http://127.0.0.1:8000/` (网站根目录)。 你将会看到一个 “Page not found” 页面（图 3－2）。 因为你的URL申请在URLconf中没有定义，所以Django显示这条信息。

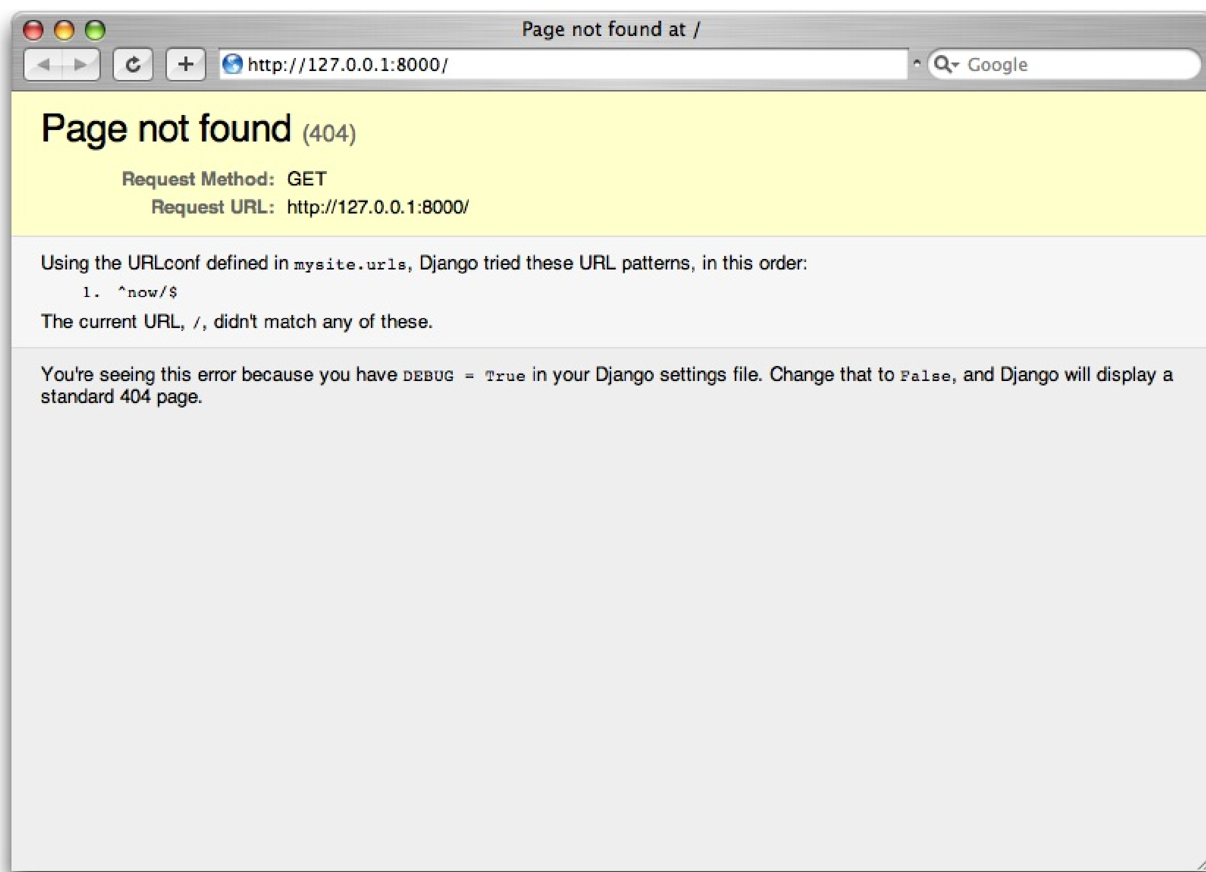


图3-1： Django的404 Error页

这个页面比原始的404错误信息更加实用。 它同时精确的告诉你Django调用哪个URLconf及其包含的每个模式。 这样，你应该能了解到为什么这个请求会抛出404错误。

当然，这些敏感的信息应该只呈现给你－开发者。 如果是部署到了因特网上的站点就不应该暴露 这些信息。 出于这个考虑，这个"Page not found"页面只会在 调试模式（*debug mode*）下 显示。 我们将在以后说明怎么关闭调试模式。

关于网站根目录的快速参考。

在最后一节，如果你想通过http://127.0.0.1:8000/看网站根目录你将看到一个404错误消息。Django不会增加任何东西在网站根目录，在任何情况下这个URL都不是特殊的 就像在URLconf中的其他条目一样，它也依赖于指定给它的URL模式.

尽管匹配网站根目录的URL模式不能想象，但是还是值得提一下的. 当为网站根目录实现一个视图，你需要使用URL模式 `'^$'` ，它代表一个空字符串。 例如:

```
from mysite.views import hello, my_homepage_view

urlpatterns = patterns('',
    ('^$', my_homepage_view),
    # ...
)
```

Django是怎么处理请求的

在继续我们的第二个视图功能之前，让我们暂停一下去了解更多一些有关Django是怎么工作的知识. 具体地说，当你通过在浏览器里敲http://127.0.0.1:8000/hello/来访问Hello world消息得时候，Django在后台有些什么动作呢？

所有均开始于setting文件。当你运行python manage.py runserver，脚本将在于manage.py同一个目录下查找名为setting.py的文件。这个文件包含了所有有关这个Django项目的配置信息，均大写： TEMPLATE_DIRS , DATABASE_NAME , 等. 最重要的设置时ROOT_URLCONF，它将作为URLconf告诉Django在这个站点中那些Python的模块将被用到

还记得什么时候django-admin.py startproject创建文件settings.py和urls.py吗？自动创建的settings.py包含一个ROOT_URLCONF配置用来指向自动产生的urls.py. 打开文件settings.py你将看到如下：

```
ROOT_URLCONF = 'mysite.urls'
```

相对应的文件是mysite/urls.py

当访问 URL `/hello/` 时，Django 根据 `ROOT_URLCONF` 的设置装载 URLconf 。 然后按顺序逐个匹配URLconf里的URLpatterns，直到找到一个匹配的。 当找到这个匹配 的URLpatterns就调用相关联的view函数，并把 `HttpRequest` 对象作为第一个参数。 （稍后再给出 `HttpRequest` 的更多信息） （我们将在后面看到HttpRequest的标准）

正如我们在第一个视图例子里面看到的，一个视图功能必须返回一个HttpResponse。 一旦做完，Django将完成剩余的转换Python的对象到一个合适的带有HTTP头和body的Web Response， （例如，网页内容）。

总结一下：

1. 进来的请求转入/hello/.
2. Django通过在ROOT_URLCONF配置来决定根URLconf.
3. Django在URLconf中的所有URL模式中，查找第一个匹配/hello/的条目。
4. 如果找到匹配，将调用相应的视图函数
5. 视图函数返回一个HttpResponse
6. Django转换HttpResponse为一个适合的HTTP response， 以Web page显示出来

你现在知道了怎么做一个 Django-powered 页面了，真的很简单，只需要写视图函数并用URLconfs把它们和URLs对应起来。 你可能会认为用一系列正则表达式将URLs映射到函数也许会比较慢，但事实却会让你惊讶。

第二个视图：动态内容

我们的Hello world视图是用来演示基本的Django是如何工作的，但是它不是一个动态网页的例子，因为网页的内容一直是一样的. 每次去查看/hello/，你将会看到相同的内容，它类似一个静态HTML文件。

我们的第二个视图，将更多的放些动态的东西例如当前日期和时间显示在网页上 这将非常好，简单的下一步，因为它不引入了数据库或者任何用户的输入，仅仅是输出显示你的服务器的内部时钟. 它仅仅有限度的比Helloworld刺激一些，但是它将演示一些新的概念

这个视图需要做两件事情：计算当前日期和时间，并返回包含这些值的HttpResponse 如果你对python很有经验，那肯定知道在python中需要利用datetime模块去计算时间 下面演示如何去使用它：

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2008, 12, 13, 14, 9, 39, 2731)
>>> print now
2008-12-13 14:09:39.002731
```

以上代码很简单，并没有涉及Django。 它仅仅是Python代码。 需要强调的是，你应该意识到哪些是纯Python代码，哪些是Django特性代码。 （见上） 因为你学习了Django，希望你能将Django的知识应用在那些不一定需要使用Django的项目上。

为了让Django视图显示当前日期和时间，我们仅需要把语句：datetime.datetime.now()放入视图函数，然后返回一个HttpResponse对象即可。代码如下：

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

正如我们的hello函数一样，这个函数也保存在view.py中。为了简洁，上面我们隐藏了hello函数。下面是完整的view.py文件内容：

```
from django.http import HttpResponse
import datetime

def hello(request):
    return HttpResponse("Hello world")

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

（从现在开始，如非必要，本文不再重复列出先前的代码。 你应该懂得识别哪些是新代码，哪些是先前的。） （见上）

让我们分析一下改动后的views.py：

> 在文件顶端，我们添加了一条语句：import datetime。这样就可以计算日期了。
>
> 函数中的第一行代码计算当前日期和时间，并以 `datetime.datetime` 对象的形式保存为局部变量 `now` 。
>
> 函数的第二行代码用 Python 的格式化字符串（format-string）功能构造了一段 HTML 响应。 字符串中的%s是占位符，字符串后面的百分号表示用它后面的变量now的值来代替%s。变量%s是一个datetime.datetime对象。它虽然不是一个字符串，但是%s（格式化字符串）会把它转换成字符串，如：2008-12-13 14:09:39.002731。这将导致HTML的输出字符串为：It is now 2008-12-13 14:09:39.002731。
>
> （目前HTML是有错误的，但我们这样做是为了保持例子的简短。）
>
> 最后，正如我们刚才写的hello函数一样，视图返回一个HttpResponse对象，它包含生成的响应。

添加上述代码之后，还要在urls.py中添加URL模式，以告诉Django由哪一个URL来处理这个视图。 用/time/之类的字眼易于理解：

```
from django.conf.urls.defaults import *
from mysite.views import hello, current_datetime

urlpatterns = patterns('',
    ('^hello/$', hello),
    ('^time/$', current_datetime),
)
```

这里，我们修改了两个地方。 首先，在顶部导入current_datetime函数； 其次，也是比较重要的：添加URL模式来映射URL中的/time/和新视图。 理解了么？

写好视图并且更新URLconf之后，运行命令python manage.py runserver以启动服务，在浏览器中输入http://127.0.0.1:8000/time/。 你将看到当前的日期和时间。

Django时区

视乎你的机器，显示的日期与时间可能和实际的相差几个小时。 这是因为Django是有时区意识的，并且默认时区为America/Chicago。 （它必须有个值，它的默认值是Django的诞生地：美国/芝加哥）如果你处在别的时区，你需要在settings.py文件中更改这个值。请参见它里面的注释，以获得最新世界时区列表。

URL配置和松耦合

现在是好时机来指出Django和URL配置背后的哲学： 松耦合 原则。 简单的说，松耦合是一个 重要的保证互换性的软件开发方法。

Django的URL配置就是一个很好的例子。 在Django的应用程序中，URL的定义和视图函数之间是松 耦合的，换句话说，决定URL返回哪个视图函数和实现这个视图函数是在两个不同的地方。 这使得 开发人员可以修改一块而不会影响另一块。

例如，考虑一下current_datetime视图。 如果我们想把它的URL 从原来的 `/time/` 改变到 `/currenttime/` ，我们只需要快速的修改一下URL配置即可， 不用担心这个函数的内部实现。 同样的，如果我们想要修改这个函数的内部实现也不用担心会影响 到对应的URL。

此外，如果我们想要输出这个函数到 一些 URL， 我们只需要修改URL配置而不用 去改动视图的代码。 在这个例子里，current_datetime被两个URL使用。 这是一个故弄玄虚的例子，但这个方法迟早会用得上。

```
urlpatterns = patterns('',
    ('^hello/$', hello),
    ('^time/$', current_datetime),
    ('^another-time-page/$', current_datetime),
)
```

URLconf和视图是松耦合的。 我们将在本书中继续给出这一重要哲学的相关例子。

第三个视图 动态URL

在我们的 `current_datetime` 视图范例中，尽管内容是动态的，但是URL （ `/time/` ）是静态的。 在 大多数动态web应用程序，URL通常都包含有相关的参数。 举个例子，一家在线书店会为每一本书提供一个URL，如：/books/243/、/books/81196/。

让我们创建第三个视图来显示当前时间和加上时间偏差量的时间，设计是这样的： `/time/plus/1/` 显示当前时间＋1个小时的页面 `/time/plus/2/` 显示当前时间＋2个小时的页面 `/time/plus/3/` 显示当前时间＋3个小时的页面，以此类推。

新手可能会考虑写不同的视图函数来处理每个时间偏差量，URL配置看起来就象这样：

```
urlpatterns = patterns('',
    ('^time/$', current_datetime),
    ('^time/plus/1/$', one_hour_ahead),
    ('^time/plus/2/$', two_hours_ahead),
    ('^time/plus/3/$', three_hours_ahead),
    ('^time/plus/4/$', four_hours_ahead),
)
```

很明显，这样处理是不太妥当的。 不但有很多冗余的视图函数，而且整个应用也被限制了只支持 预先定义好的时间段，2小时，3小时，或者4小时。 如果哪天我们要实现 *5* 小时，我们就 不得不再单独创建新的视图函数和配置URL，既重复又混乱。 我们需要在这里做一点抽象，提取 一些共同的东西出来。

关于漂亮URL的一点建议

如果你有其它web平台的开发经验（如PHP或Java），你可能会想：嘿！让我们用查询字符串参数吧！ 就像/time/plus?hours=3里面的小时应该在查询字符串中被参数hours指定（问号后面的是参数）。

你 可以 在Django里也这样做 (如果你真的想要这样做，我们稍后会告诉你怎么做)， 但是Django的一个核心理念就是URL必须看起来漂亮。 URL `/time/plus/3/` 更加清晰， 更简单，也更有可读性，可以很容易的大声念出来，因为它是纯文本，没有查询字符串那么 复杂。 漂亮的URL就像是高质量的Web应用的一个标志。

Django的URL配置系统可以使你很容易的设置漂亮的URL，而尽量不要考虑它的 反面 。

那么，我们如何设计程序来处理任意数量的时差？ 答案是：使用通配符（wildcard URLpatterns）。正如我们之前提到过，一个URL模式就是一个正则表达式。因此，这里可以使用d+来匹配1个以上的数字。

```
urlpatterns = patterns('',
    # ...
    (r'^time/plus/\d+/$', hours_ahead),
    # ...
)
```

这里使用# …来表示省略了其它可能存在的URL模式定义。 （见上）

这个URL模式将匹配类似 `/time/plus/2/` , `/time/plus/25/` ,甚至 `/time/plus/100000000000/` 的任何URL。 更进一步，让我们把它限制在最大允许99个小时， 这样我们就只允许一个或两个数字，正则表达式的语法就是 `\d{1,2}` :

```
(r'^time/plus/\d{1,2}/$', hours_ahead),
```

备注

在建造Web应用的时候，尽可能多考虑可能的数据输入是很重要的，然后决定哪些我们可以接受。 在这里我们就设置了99个小时的时间段限制。

另外一个重点，正则表达式字符串的开头字母“r”。 它告诉Python这是个原始字符串，不需要处理里面的反斜杠（转义字符）。 在普通Python字符串中，反斜杠用于特殊字符的转义。比如n转义成一个换行符。 当你用r把它标示为一个原始字符串后，Python不再视其中的反斜杠为转义字符。也就是说，“n”是两个字符串：“”和“n”。由于反斜杠在Python代码和正则表达式中有冲突，因此建议你在Python定义正则表达式时都使用原始字符串。 从现在开始，本文所有URL模式都用原始字符串。

现在我们已经设计了一个带通配符的URL，我们需要一个方法把它传递到视图函数里去，这样 我们只用一个视图函数就可以处理所有的时间段了。 我们使用圆括号把参数在URL模式里标识 出来。 在这个例子中，我们想要把这些数字作为参数，用圆括号把 `\d{1,2}` 包围起

来：

```
(r'^time/plus/(\d{1,2})/$', hours_ahead),
```

如果你熟悉正则表达式，那么你应该已经了解，正则表达式也是用圆括号来从文本里 提取 数据的。

最终的URLconf包含上面两个视图，如：

```
from django.conf.urls.defaults import *
from mysite.views import hello, current_datetime, hours_ahead

urlpatterns = patterns('',
    (r'^hello/$', hello),
    (r'^time/$', current_datetime),
    (r'^time/plus/(\d{1,2})/$', hours_ahead),
)
```

现在开始写 `hours_ahead` 视图。

编码次序

这个例子中，我们先写了URLpattern ，然后是视图，但是在前面的例子中， 我们先写了视图，然后是URLpattern 。 哪一种方式比较好？

嗯，怎么说呢，每个开发者是不一样的。

如果你是喜欢从总体上来把握事物（注： 或译为“大局观”）类型的人，你应该会想在项目开始 的时候就写下所有的URL配置。

如果你从更像是一个自底向上的开发者，你可能更喜欢先写视图， 然后把它们挂接到URL上。 这同样是可以的。

最后，取决与你喜欢哪种技术，两种方法都是可以的。 (见上)

`hours_ahead` 和我们以前写的 `current_datetime` 很象，关键的区别在于： 它多了一个额外参数，时间差。 以下是view代码：

```
from django.http import Http404, HttpResponse
import datetime

def hours_ahead(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

让我们逐行分析一下代码：

> 视图函数, `hours_ahead` , 有 两个 参数: `request` 和 `offset` . (见上)
>
> > `request` 是一个 `HttpRequest` 对象, 就像在 `current_datetime` 中一样. 再说一次好了: 每一个视图 总是 以一个 `HttpRequest` 对象作为 它的第一个参数。 (见上)
> >
> > `offset` 是从匹配的URL里提取出来的。 例如：如果请求URL是/time/plus/3/，那么offset将会是3；如果请求URL是/time/plus/21/，那么offset将会是21。请注意：捕获值永远都是字符串（string）类型，而不会是整数（integer）类型，即使这个字符串全由数字构成（如：“21”）。
> >
> > （从技术上来说，捕获值总是Unicode objects，而不是简单的Python字节串，但目前不需要担心这些差别。）
> >
> > 在这里我们命名变量为 `offset` ，你也可以任意命名它，只要符合Python 的语法。变量名是无关紧要的，重要的是它的位置，它是这个函数的第二个 参数 (在 `request` 的后面）。 你还可以使用关键字来定义它，而不是用 位置。
>
> 我们在这个函数中要做的第一件事情就是在 `offset` 上调用 `int()` . 这会把这个字符串值转换为整数。
>
> 请留意：如果你在一个不能转换成整数类型的值上调用int()，Python将抛出一个ValueError异常。如：int(‘foo’)。在这个例子中，如果我们遇到ValueError异常，我们将转为抛出django.http.Http404异常——正如你想象的那样：最终显示404页面（提示信息：页面不存在）。
>
> 机灵的读者可能会问： 我们在URL模式中用正则表达式(d{1,2})约束它，仅接受数字怎么样？这样无论如何，offset都是由数字构成的。 答案是：我们不会这么做，因为URLpattern提供的是“适度但有用”级别的输入校验。万一这个视图函数被其它方式调用，我们仍需自行检查ValueError。 实践证明，在实现视图函数时，不臆测参数值的做法是比较好的。 松散耦合，还记得么？
>
> 下一行，计算当前日期/时间，然后加上适当的小时数。 在current_datetime视图中，我们已经见过datetime.datetime.now()。这里新的概念是执行日期/时间的算术操作。我们需要创建一个datetime.timedelta对象和增加一个datetime.datetime对象。 结果保存在变量dt中。
>
> 这一行还说明了，我们为什么在offset上调用int()——datetime.timedelta函数要求hours参数必须为整数类型。
>
> 这行和前面的那行的的一个微小差别就是，它使用带有两个值的Python的格式化字符串功能， 而不仅仅是一个值。 因此，在字符串中有两个 `%s` 符号和一个以进行插入的值的元组： `(offset, dt)` 。
>
> 最终，返回一个HTML的HttpResponse。 如今，这种方式已经过时了。

在完成视图函数和URL配置编写后，启动Django开发服务器，用浏览器访问 `http://127.0.0.1:8000/time/plus/3/` 来确认它工作正常。 然后是 `http://127.0.0.1:8000/time/plus/5/` 。再然后是 `http://127.0.0.1:8000/time/plus/24/` 。最后，访问 `http://127.0.0.1:8000/time/plus/100/` 来检验URL配置里设置的模式是否只 接受一个或两个数字；Django会显示一个 Page not found error 页面, 和以前看到的 404 错误一样。访问URL `http://127.0.0.1:8000/time/plus/` (没有 定义时间差) 也会抛出404错误。

Django 漂亮的出错页面

花几分钟时间欣赏一下我们写好的Web应用程序，然后我们再来搞点小破坏。 我们故意在 `views.py` 文件中引入一项 Python 错误，注释掉 `hours_ahead` 视图中的 `offset = int(offset)` 一行。

```
def hours_ahead(request, offset):
    # try:
    #     offset = int(offset)
    # except ValueError:
    #     raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

启动开发服务器，然后访问 `/time/plus/3/` 。你会看到一个包含大量信息的出错页，最上面的一条 `TypeError` 信息是： `"unsupported type for timedelta hours component: unicode"` .

怎么回事呢？ 是的， `datetime.timedelta` 函数要求 `hours` 参数必须为整型， 而我们注释掉了将 `offset` 转为整型的代码。 这样导致 `datetime.timedelta` 弹出 `TypeError` 异常。

这个例子是为了展示 Django 的出错页面。 我们来花些时间看一看这个出错页，了解一下其中 给出了哪些信息。

以下是值得注意的一些要点：

> 在页面顶部，你可以得到关键的异常信息： 异常数据类型、异常的参数 (如本例中的 `"unsupported type"` ）、在哪个文件中引发了异常、出错的行号等等。
>
> 在关键异常信息下方，该页面显示了对该异常的完整 Python 追踪信息。 这类似于你在 Python 命令行解释器中获得的追溯信息，只不过后者更具交互性。 对栈中的每一帧，Django 均显示了其文件名、函数或方法名、行号及该行源代码。
>
> 点击该行代码 (以深灰色显示)，你可以看到出错行的前后几行，从而得知相关上下文情况。
>
> 点击栈中的任何一帧的“Local vars”可以看到一个所有局部变量的列表，以及在出错 那一帧时它们的值。 这些调试信息相当有用。
>
> 注意“Traceback”下面的“Switch to copy-and-paste view”文字。 点击这些字，追溯会 切换另一个视图，它让你很容易地复制和粘贴这些内容。 当你想同其他人分享这些异常 追溯以获得技术支持时（比如在 Django 的 IRC 聊天室或邮件列表中），可以使用它。
>
> 你按一下下面的“Share this traceback on a public Web site”按钮，它将会完成这项工作。 点击它以传回追溯信息至http://www.dpaste.com/，在那里你可以得到一个单独的URL并与其他人分享你的追溯信息。
>
> 接下来的“Request information”部分包含了有关产生错误的 Web 请求的大量信息： GET 和 POST、cookie 值、元数据（象 CGI 头）。 在附录H里给出了request的对象的 完整参考。
>
> Request信息的下面，“Settings”列出了 Django 使用的具体配置信息。 （我们已经提及过ROOT_URLCONF，接下来我们将向你展示各式的Django设置。 附录D覆盖了所有可用的设置。）

Django 的出错页某些情况下有能力显示更多的信息，比如模板语法错误。 我们讨论 Django 模板系统时再说它们。 现在，取消 `offset = int(offset)` 这行的注释，让它重新正常 工作。

不知道你是不是那种使用小心放置的 `print` 语句来帮助调试的程序员？ 你其实可以用 Django 出错页来做这些，而不用 `print` 语句。 在你视图的任何位置，临时插入一个 `assert False` 来触发出错页。 然后，你就可以看到局部变量和程序语句了。 这里有个使用 hours_ahead视图的例子：

```
def hours_ahead(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    assert False
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

最后，很显然这些信息很多是敏感的，它暴露了你 Python 代码的内部结构以及 Django 配置，在 Internet 上公开这信息是很愚蠢的。 不怀好意的人会尝试使用它攻击你的 Web 应用程序，做些下流之事。 因此，Django 出错信息仅在 debug 模式下才会显现。 我们稍后 说明如何禁用 debug 模式。 现在，你只要知道 Django 服务器在你开启它时默认运行在 debug 模式就行了。 （听起来很熟悉？ 页面没有发现错误，如前所述，工作正常。）

下一章

目前为止，我们已经写好了视图函数和硬编码的HTML。 在演示核心概念时，我们所作的是为了保持简单。但是在现实世界中，这差不多总是个坏主意。

幸运的是，Django内建有一个简单有强大的模板处理引擎来让你分离两种工作： 下一章，我们将学习模板引擎。

第四章：模板

在前一章中，你可能已经注意到我们在例子视图中返回文本的方式有点特别。 也就是说，HTML被直接硬编码在 Python 代码之中。

```
def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

尽管这种技术便于解释视图是如何工作的，但直接将HTML硬编码到你的视图里却并不是一个好主意。 让我们来看一下为什么：

- 对页面设计进行的任何改变都必须对 Python 代码进行相应的修改。 站点设计的修改往往比底层 Python 代码的修改要频繁得多，因此如果可以在不进行 Python 代码修改的情况下变更设计，那将会方便得多。
- Python 代码编写和 HTML 设计是两项不同的工作，大多数专业的网站开发环境都将他们分配给不同的人员（甚至不同部门）来完成。 设计者和HTML/CSS的编码人员不应该被要求去编辑Python的代码来完成他们的工作。
- 程序员编写 Python代码和设计人员制作模板两项工作同时进行的效率是最高的，远胜于让一个人等待另一个人完成对某个既包含 Python又包含 HTML 的文件的编辑工作。

基于这些原因，将页面的设计和Python的代码分离开会更干净简洁更容易维护。 我们可以使用 Django的 模板系统 (Template System)来实现这种模式，这就是本章要具体讨论的问题。

模板系统基本知识

模板是一个文本，用于分离文档的表现形式和内容。 模板定义了占位符以及各种用于规范文档该如何显示的各部分基本逻辑（模板标签）。 模板通常用于产生HTML，但是Django的模板也能产生任何基于文本格式的文档。

让我们从一个简单的例子模板开始。 该模板描述了一个向某个与公司签单人员致谢 HTML 页面。 可将其视为一个格式信函：

```
<html>
<head><title>Ordering notice</title></head>

<body>

<h1>Ordering notice</h1>

<p>Dear {{ person_name }},</p>

<p>Thanks for placing an order from {{ company }}. It's scheduled to
ship on {{ ship_date|date:"F j, Y" }}.</p>

<p>Here are the items you've ordered:</p>

<ul>
{% for item in item_list %}
    <li>{{ item }}</li>
{% endfor %}
</ul>

{% if ordered_warranty %}
    <p>Your warranty information will be included in the packaging.</p>
{% else %}
    <p>You didn't order a warranty, so you're on your own when
    the products inevitably stop working.</p>
{% endif %}

<p>Sincerely,<br />{{ company }}</p>

</body>
</html>
```

该模板是一段添加了些许变量和模板标签的基础 HTML 。 让我们逐步分析一下：

> 用两个大括号括起来的文字（例如 `{{ person_name }}` ）称为 变量*(variable)* 。这意味着在此处插入指定变量的值。 如何指定变量的值呢？ 稍后就会说明。
>
> 被大括号和百分号包围的文本(例如 `{% if ordered_warranty %}`)是 模板标签*(template tag)* 。标签(tag)定义比较明确，即： 仅通知模板系统完成某些工作的标签。
>
> 这个例子中的模板包含一个`for`标签（ `{% for item in item_list %}` ）和一个`if` 标签（`{% if ordered_warranty %}` ）
>
> for标签类似Python的for语句，可让你循环访问序列里的每一个项目。 `if` 标签，正如你所料，是用来执行逻辑判断的。 在这里，tag标签检查ordered_warranty值是否为True。如果是，模板系统将显示{% if ordered_warranty %}和{% else %}之间的内容；否则将显示{% else %}和{% endif %}之间的内容。{% else %}是可选的。
>
> 最后，这个模板的第二段中有一个关于*filter*过滤器的例子，它是一种最便捷的转换变量输出格式的方式。 如这个例子中的{{ship_date|date:”F j, Y” }}，我们将变量ship_date传递给date过滤器，同时指定参数”F j,Y”。date过滤器根据参数进行格式输出。 过滤器是用管道符(|)来调用的，具体可以参见Unix管道符。

Django 模板含有很多内置的tags和filters,我们将陆续进行学习. 附录F列出了很多的tags和filters的列表,熟悉这些列表对你来说是个好建议. 你依然可以利用它创建自己的tag和filters。这些我们在第9章会讲到。

如何使用模板系统

让我们深入研究模板系统，你将会明白它是如何工作的。但我们暂不打算将它与先前创建的视图结合在一起，因为我们现在的目的是了解它是如何独立工作的。 。 （换言之， 通常你会将模板和视图一起使用，但是我们只是想突出模板系统是一个Python库，你可以在任何地方使用它，而不仅仅是在Django视图中。）

在Python代码中使用Django模板的最基本方式如下：

1. 可以用原始的模板代码字符串创建一个 `Template` 对象， Django同样支持用指定模板文件路径的方式来创建 `Template` 对象;
2. 调用模板对象的render方法，并且传入一套变量context。它将返回一个基于模板的展现字符串，模板中的变量和标签会被context值替换。

代码如下：

```
>>> from django import template
>>> t = template.Template('My name is {{ name }}.')
>>> c = template.Context({'name': 'Adrian'})
>>> print t.render(c)
My name is Adrian.
>>> c = template.Context({'name': 'Fred'})
>>> print t.render(c)
My name is Fred.
```

以下部分逐步的详细介绍

创建模板对象

创建一个 `Template` 对象最简单的方法就是直接实例化它。 `Template` 类就在 `django.template` 模块中，构造函数接受一个参数，原始模板代码。 让我们深入挖掘一下Python的解释器看看它是怎么工作的。

转到project目录（在第二章由 `django-admin.py startproject` 命令创建）， 输入命令 `python manage.py shell` 启动交互界面。

一个特殊的Python提示符

如果你曾经使用过Python，你一定好奇，为什么我们运行 `python manage.py shell` 而不是 `python` 。这两个命令都会启动交互解释器，但是 `manage.py shell` 命令有一个重要的不同： 在启动解释器之前，它告诉Django使用哪个设置文件。 Django框架的大部分子系统，包括模板系统，都依赖于配置文件；如果Django不知道使用哪个配置文件，这些系统将不能工作。

如果你想知道，这里将向你解释它背后是如何工作的。 Django搜索DJANGO_SETTINGS_MODULE环境变量，它被设置在settings.py中。例如，假设mysite在你的Python搜索路径中，那么DJANGO_SETTINGS_MODULE应该被设置为：'mysite.settings'。

当你运行命令：python manage.py shell，它将自动帮你处理DJANGO_SETTINGS_MODULE。 在当前的这些示例中，我们鼓励你使用 `python manage.py shell` 这个方法，这样可以免去你大费周章地去配置那些你不熟悉的环境变量。

随着你越来越熟悉Django，你可能会偏向于废弃使用 `manage.py shell` ，而是在你的配置文件 `.bash_profile` 中手动添加 `DJANGO_SETTINGS_MODULE` 这个环境变量。

让我们来了解一些模板系统的基本知识：

```
>>> from django.template import Template
>>> t = Template('My name is {{ name }}.')
>>> print t
```

如果你跟我们一起做，你将会看到下面的内容：

```
<django.template.Template object at 0xb7d5f24c>
```

`0xb7d5f24c` 每次都会不一样，这没什么关系；这只是Python运行时 `Template` 对象的ID。

当你创建一个 `Template` 对象，模板系统在内部编译这个模板到内部格式，并做优化，做好渲染的准备。 如果你的模板语法有错误，那么在调用 `Template()` 时就会抛出 `TemplateSyntaxError` 异常：

```
>>> from django.template import Template
>>> t = Template('{% notatag %}')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  ...
django.template.TemplateSyntaxError: Invalid block tag: 'notatag'
```

这里，块标签(block tag)指向的是 `{% notatag %}` ，块标签与模板标签是同义的。

系统会在下面的情形抛出 `TemplateSyntaxError` 异常：

- 无效的tags
- 标签的参数无效
- 无效的过滤器
- 过滤器的参数无效

- 无效的模板语法
- 未封闭的块标签 （针对需要封闭的块标签）

模板渲染

一旦你创建一个 `Template` 对象，你可以用 *context* 来传递数据给它。 一个context是一系列变量和它们值的集合。

context在Django里表现为 `Context` 类，在 `django.template` 模块里。 她的构造函数带有一个可选的参数： 一个字典映射变量和它们的值。 调用 `Template` 对象 的 `render()` 方法并传递context来填充模板：

```
>>> from django.template import Context, Template
>>> t = Template('My name is {{ name }}.')
>>> c = Context({'name': 'Stephane'})
>>> t.render(c)
u'My name is Stephane.'
```

我们必须指出的一点是， `t.render(c)` 返回的值是一个Unicode对象，不是普通的Python字符串。 你可以通过字符串前的 `u` 来区分。 在框架中，Django会一直使用Unicode对象而不是普通的字符串。 如果你明白这样做给你带来了多大便利的话，尽可能地感激Django在幕后有条不紊地为你所做这这么多工作吧。 如果不明白你从中获益了什么， 别担心。你只需要知道Django对Unicode的支持，将让你的应用程序轻松地处理各式各样的字符集，而不仅仅是基本的A-Z英文字符。

字典和Contexts

Python的字典数据类型就是关键字和它们值的一个映射。 `Context` 和字典很类似， `Context` 还提供更多的功能，请看第九章。

变量名必须由英文字符开始 （A-Z或a-z）并可以包含数字字符、下划线和小数点。 （小数点在这里有特别的用途，稍后我们会讲到）变量是大小写敏感的。

下面是编写模板并渲染的示例：

```
>>> from django.template import Template, Context
>>> raw_template = """<p>Dear {{ person_name }},</p>
...
... <p>Thanks for placing an order from {{ company }}. It's scheduled to
... ship on {{ ship_date|date:"F j, Y" }}.</p>
...
... {% if ordered_warranty %}
... <p>Your warranty information will be included in the packaging.</p>
... {% else %}
... <p>You didn't order a warranty, so you're on your own when
... the products inevitably stop working.</p>
... {% endif %}
...
... <p>Sincerely,<br />{{ company }}</p>"""
>>> t = Template(raw_template)
>>> import datetime
>>> c = Context({'person_name': 'John Smith',
...     'company': 'Outdoor Equipment',
...     'ship_date': datetime.date(2009, 4, 2),
...     'ordered_warranty': False})
>>> t.render(c)
u"<p>Dear John Smith,</p>\n\n<p>Thanks for placing an order from Outdoor
Equipment. It's scheduled to\nship on April 2, 2009.</p>\n\n\n<p>You
didn't order a warranty, so you're on your own when\nthe products
inevitably stop working.</p>\n\n\n<p>Sincerely,<br />Outdoor Equipment
</p>"
```

让我们逐步来分析下这段代码：

> 首先我们导入（import）类 `Template` 和 `Context` ，它们都在模块 `django.template` 里。
>
> 我们把模板原始文本保存到变量 `raw_template` 。注意到我们使用了三个引号来 标识这些文本，因为这样可以包含多行。
>
> 接下来，我们创建了一个模板对象 `t` ，把 `raw_template` 作为 `Template` 类构造函数的参数。
>
> 我们从Python的标准库导入 `datetime` 模块，以后我们将会使用它。
>
> 然后，我们创建一个 `Context` 对象， `c` 。 `Context` 构造的参数是Python 字典数据类型。 在这里，我们指定参数 `person_name` 的值是 `'John Smith'` ，参数company 的值为 ‘Outdoor Equipment’， 等等。
>
> 最后，我们在模板对象上调用 `render()` 方法，传递 context参数给它。 这是返回渲染后的模板的方法，它会替换模板变量为真实的值和执行块标签。
>
> 注意，warranty paragraph显示是因为 `ordered_warranty` 的值为 `True` . 注意时间的显示， `April 2, 2009` ，它是按 `'F j, Y'` 格式显示的。
>
> 如果你是Python初学者，你可能在想为什么输出里有回车换行的字符(`'\n'`)而不是 显示回车换行？因为这是Python交互解释器的缘故： 调用 `t.render(c)` 返回字符串， 解释器缺省显示这些字符串的 真实内容呈现 ，而不是打印这个变量的值。 要显示换行而不是 `'\n'` ，使用 `print` 语句： `print t.render(c)` 。

这就是使用Django模板系统的基本规则：写模板，创建 `Template` 对象，创建 `Context` ，调用 `render()` 方法。

同一模板，多个上下文

一旦有了 `模板` 对象，你就可以通过它渲染多个context，例如：

```
>>> from django.template import Template, Context
>>> t = Template('Hello, {{ name }}')
>>> print t.render(Context({'name': 'John'}))
Hello, John
>>> print t.render(Context({'name': 'Julie'}))
Hello, Julie
>>> print t.render(Context({'name': 'Pat'}))
Hello, Pat
```

无论何时我们都可以像这样使用同一模板源渲染多个context，只进行 `一次` 模板创建然后多次调用render()方法渲染会更为高效：

```
# Bad
for name in ('John', 'Julie', 'Pat'):
    t = Template('Hello, {{ name }}')
    print t.render(Context({'name': name}))

# Good
t = Template('Hello, {{ name }}')
for name in ('John', 'Julie', 'Pat'):
    print t.render(Context({'name': name}))
```

Django 模板解析非常快捷。 大部分的解析工作都是在后台通过对简短正则表达式一次性调用来完成。 这和基于 XML 的模板引擎形成鲜明对比，那些引擎承担了 XML 解析器的开销，且往往比 Django 模板渲染引擎要慢上几个数量级。

深度变量的查找

在到目前为止的例子中，我们通过 context 传递的简单参数值主要是字符串，还有一个 `datetime.date` 范例。 然而，模板系统能够非常简洁地处理更加复杂的数据结构，例如list、dictionary和自定义的对象。

在 Django 模板中遍历复杂数据结构的关键是句点字符 (`.`)。

最好是用几个例子来说明一下。 比如，假设你要向模板传递一个 Python 字典。 要通过字典键访问该字典的值，可使用一个句点：

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
u'Sally is 43 years old.'
```

同样，也可以通过句点来访问对象的属性。 比方说， Python 的 `datetime.date` 对象有 `year` 、 `month` 和 `day` 几个属性，你同样可以在模板中使用句点来访问这些属性：

```
>>> from django.template import Template, Context
>>> import datetime
>>> d = datetime.date(1993, 5, 2)
>>> d.year
1993
>>> d.month
5
>>> d.day
2
>>> t = Template('The month is {{ date.month }} and the year is {{ date.year }}.')
>>> c = Context({'date': d})
>>> t.render(c)
u'The month is 5 and the year is 1993.'
```

这个例子使用了一个自定义的类，演示了通过实例变量加一点(dots)来访问它的属性，这个方法适用于任意的对象。

```
>>> from django.template import Template, Context
>>> class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name, self.last_name = first_name, last_name
>>> t = Template('Hello, {{ person.first_name }} {{ person.last_name }}.')
>>> c = Context({'person': Person('John', 'Smith')})
>>> t.render(c)
u'Hello, John Smith.'
```

点语法也可以用来引用对象的 方法。 例如，每个 Python 字符串都有 `upper()` 和 `isdigit()` 方法，你在模板中可以使用同样的句点语法来调用它们：

```
>>> from django.template import Template, Context
>>> t = Template('{{ var }} -- {{ var.upper }} -- {{ var.isdigit }}')
>>> t.render(Context({'var': 'hello'}))
u'hello -- HELLO -- False'
>>> t.render(Context({'var': '123'}))
u'123 -- 123 -- True'
```

注意这里调用方法时并 没有 使用圆括号 而且也无法给该方法传递参数；你只能调用不需参数的方法。 （我们将在本章稍后部分解释该设计观。）

最后，句点也可用于访问列表索引，例如：

```
>>> from django.template import Template, Context
>>> t = Template('Item 2 is {{ items.2 }}.')
>>> c = Context({'items': ['apples', 'bananas', 'carrots']})
>>> t.render(c)
u'Item 2 is carrots.'
```

不允许使用负数列表索引。 像 `{{ items.-1 }}` 这样的模板变量将会引发 `TemplateSyntaxError`

Python 列表类型

一点提示： Python的列表是从0开始索引。 第一项的索引是0，第二项的是1，依此类推。

句点查找规则可概括为： 当模板系统在变量名中遇到点时，按照以下顺序尝试进行查找：

- 字典类型查找 （比如 `foo["bar"]` ）
- 属性查找 (比如 `foo.bar`)
- 方法调用 （比如 `foo.bar()` ）
- 列表类型索引查找 (比如 `foo[bar]`)

系统使用找到的第一个有效类型。 这是一种短路逻辑。

句点查找可以多级深度嵌套。 例如在下面这个例子中 `{{person.name.upper}}` 会转换成字典类型查找（ `person['name']`) 然后是方法调用（ `upper()`):

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name.upper }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
u'SALLY is 43 years old.'
```

方法调用行为

方法调用比其他类型的查找略为复杂一点。 以下是一些注意事项：

> 在方法查找过程中，如果某方法抛出一个异常，除非该异常有一个 `silent_variable_failure` 属性并且值为 `True` ，否则的话它将被传播。如果异常被传播，模板里的指定变量会被置为空字符串，比如:

```
>>> t = Template("My name is {{ person.first_name }}.")
>>> class PersonClass3:
...     def first_name(self):
...         raise AssertionError, "foo"
>>> p = PersonClass3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo

>>> class SilentAssertionError(AssertionError):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
u'My name is .'
```

> 仅在方法无需传入参数时，其调用才有效。 否则，系统将会转移到下一个查找类型（列表索引查找）。
>
> 显然，有些方法是有副作用的，好的情况下允许模板系统访问它们可能只是干件蠢事，坏的情况下甚至会引发安全漏洞。
>
> 例如，你的一个 `BankAccount` 对象有一个 `delete()` 方法。 如果某个模板中包含了像 `{{ account.delete }}`这样的标签，其中`account` 又是`BankAccount` 的一个实例，请注意在这个模板载入时，account对象将被删除。
>
> 要防止这样的事情发生，必须设置该方法的 `alters_data` 函数属性：

```
def delete(self):
    # Delete the account
delete.alters_data = True
```

> 模板系统不会执行任何以该方式进行标记的方法。 接上面的例子，如果模板文件里包含了 `{{ account.delete }}` ，对象又具有 `delete()`方法，而且`delete()`有`alters_data=True`这个属性，那么在模板载入时， `delete()`方法将不会被执行。 它将静静地错误退出。

如何处理无效变量

默认情况下，如果一个变量不存在，模板系统会把它展示为空字符串，不做任何事情来表示失败。 例如：

```
>>> from django.template import Template, Context
>>> t = Template('Your name is {{ name }}.')
>>> t.render(Context())
u'Your name is .'
>>> t.render(Context({'var': 'hello'}))
u'Your name is .'
>>> t.render(Context({'NAME': 'hello'}))
u'Your name is .'
>>> t.render(Context({'Name': 'hello'}))
u'Your name is .'
```

系统静悄悄地表示失败，而不是引发一个异常，因为这通常是人为错误造成的。 这种情况下，因为变量名有错误的状况或名称， 所有的查询都会失败。 现实世界中，对于一个web站点来说，如果仅仅因为一个小的模板语法错误而造成无法访问，这是不可接受的。

玩一玩上下文(context)对象

多数时间，你可以通过传递一个完全填充(full populated)的字典给 `Context()` 来初始化 `上下文(Context)` 。 但是初始化以后，你也可以使用标准的Python字典语法(syntax)向 `上下文(Context)` 对象添加或者删除条目:

```
>>> from django.template import Context
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
Traceback (most recent call last):
  ...
KeyError: 'foo'
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

基本的模板标签和过滤器

像我们以前提到过的，模板系统带有内置的标签和过滤器。 下面的章节提供了一个多数通用标签和过滤器的简要说明。

标签

if/else

`{% if %}` 标签检查(evaluate)一个变量，如果这个变量为真（即，变量存在，非空，不是布尔值假），系统会显示在 `{% if %}` 和 `{% endif %}` 之间的任何内容，例如：

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% endif %}
```

`{% else %}` 标签是可选的：

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% else %}
    <p>Get back to work.</p>
{% endif %}
```

Python 的“真值”

在Python和Django模板系统中，以下这些对象相当于布尔值的 `False`

- 空列表(`[]`)
- 空元组(`()`)
- 空字典(`{}`)
- 空字符串(`''`)
- 零值(`0`)

- 特殊对象 `None`
- 对象 `False` （很明显）
- 提示：你也可以在自定义的对象里定义他们的布尔值属性(这个是python的高级用法)。

除以上几点以外的所有东西都视为 `True`

`{% if %}` 标签接受 `and` ， `or` 或者 `not` 关键字来对多个变量做判断 ，或者对变量取反（ `not`)，例如： 例如：

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
    There are no athletes or there are some coaches.
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% endif %}
```

`{% if %}` 标签不允许在同一个标签中同时使用 `and` 和 `or` ，因为逻辑上可能模糊的，例如，如下示例是错误的： 比如这样的代码是不合法的：

```
{% if athlete_list and coach_list or cheerleader_list %}
```

系统不支持用圆括号来组合比较操作。 如果你确实需要用到圆括号来组合表达你的逻辑式，考虑将它移到模板之外处理，然后以模板变量的形式传入结果吧。 或者，仅仅用嵌套的 `{% if %}` 标签替换吧，就像这样：

```
{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        We have athletes, and either coaches or cheerleaders!
    {% endif %}
{% endif %}
```

多次使用同一个逻辑操作符是没有问题的，但是我们不能把不同的操作符组合起来。 例如，这是合法的：

```
{% if athlete_list or coach_list or parent_list or teacher_list %}
```

并没有 `{% elif %}` 标签， 请使用嵌套的 `{% if %}` 标签来达成同样的效果：

```
{% if athlete_list %}
    <p>Here are the athletes: {{ athlete_list }}.</p>
{% else %}
    <p>No athletes are available.</p>
    {% if coach_list %}
        <p>Here are the coaches: {{ coach_list }}.</p>
    {% endif %}
{% endif %}
```

一定要用 `{% endif %}` 关闭每一个 `{% if %}` 标签。

for

`{% for %}` 允许我们在一个序列上迭代。 与Python的 `for` 语句的情形类似，循环语法是 `for X in Y` ，Y是要迭代的序列而X是在每一个特定的循环中使用的变量名称。 每一次循环中，模板系统会渲染在 `{% for %}` 和 `{% endfor %}` 之间的所有内容。

例如，给定一个运动员列表 `athlete_list` 变量，我们可以使用下面的代码来显示这个列表：

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

给标签增加一个 `reversed` 使得该列表被反向迭代：

```
{% for athlete in athlete_list reversed %}
...
{% endfor %}
```

可以嵌套使用 `{% for %}` 标签：

```
{% for athlete in athlete_list %}
    <h1>{{ athlete.name }}</h1>
    <ul>
    {% for sport in athlete.sports_played %}
        <li>{{ sport }}</li>
    {% endfor %}
    </ul>
{% endfor %}
```

在执行循环之前先检测列表的大小是一个通常的做法，当列表为空时输出一些特别的提示。

```
{% if athlete_list %}
    {% for athlete in athlete_list %}
        <p>{{ athlete.name }}</p>
    {% endfor %}
{% else %}
    <p>There are no athletes. Only computer programmers.</p>
{% endif %}
```

因为这种做法十分常见，所以 `for` 标签支持一个可选的 `{% empty %}` 分句，通过它我们可以定义当列表为空时的输出内容 下面的例子与之前那个等价：

```
{% for athlete in athlete_list %}
    <p>{{ athlete.name }}</p>
{% empty %}
    <p>There are no athletes. Only computer programmers.</p>
{% endfor %}
```

Django不支持退出循环操作。 如果我们想退出循环，可以改变正在迭代的变量，让其仅仅包含需要迭代的项目。 同理，Django也不支持continue语句，我们无法让当前迭代操作跳回到循环头部。 （请参看本章稍后的理念和限制小节，了解下决定这个设计的背后原因）

在每个 `{% for %}` 循环里有一个称为 `forloop` 的模板变量。这个变量有一些提示循环进度信息的属性。

> `forloop.counter` 总是一个表示当前循环的执行次数的整数计数器。 这个计数器是从1开始的，所以在第一次循环时 `forloop.counter` 将会被设置为1。

```
{% for item in todo_list %}
    <p>{{ forloop.counter }}: {{ item }}</p>
{% endfor %}
```

> `forloop.counter0` 类似于 `forloop.counter` ，但是它是从0计数的。 第一次执行循环时这个变量会被设置为0。
>
> `forloop.revcounter` 是表示循环中剩余项的整型变量。 在循环初次执行时 `forloop.revcounter` 将被设置为序列中项的总数。 最后一次循环执行中，这个变量将被置1。
>
> `forloop.revcounter0` 类似于 `forloop.revcounter` ，但它以0做为结束索引。 在第一次执行循环时，该变量会被置为序列的项的个数减1。
>
> `forloop.first` 是一个布尔值，如果该迭代是第一次执行，那么它被置为```` 在下面的情形中这个变量是很有用的：
>
> System Message: WARNING/2 (`<string>` , line 1071); *backlink*
>
> Inline literal start-string without end-string.

```
{% for object in objects %}
    {% if forloop.first %}<li class="first">{% else %}<li>{% endif %}
    {{ object }}
    </li>
{% endfor %}
```

> `forloop.last` 是一个布尔值；在最后一次执行循环时被置为True。 一个常见的用法是在一系列的链接之间放置管道符（|）

```
{% for link in links %}{{ link }}{% if not forloop.last %} | {% endif %}{% endfor %}
```

> 上面的模板可能会产生如下的结果：

```
Link1 | Link2 | Link3 | Link4
```

> 另一个常见的用途是为列表的每个单词的加上逗号。

```
Favorite places:
{% for p in places %}{{ p }}{% if not forloop.last %}, {% endif %}{% endfor %}
```

> `forloop.parentloop` 是一个指向当前循环的上一级循环的 `forloop` 对象的引用（在嵌套循环的情况下）。 例子在此：

```
{% for country in countries %}
    <table>
    {% for city in country.city_list %}
        <tr>
        <td>Country #{{ forloop.parentloop.counter }}</td>
        <td>City #{{ forloop.counter }}</td>
        <td>{{ city }}</td>
        </tr>
    {% endfor %}
    </table>
{% endfor %}
```

`forloop` 变量仅仅能够在循环中使用。 在模板解析器碰到 `{% endfor %}` 标签后， `forloop` 就不可访问了。

Context和forloop变量

在一个 `{% for %}` 块中，已存在的变量会被移除，以避免 `forloop` 变量被覆盖。 Django会把这个变量移动到 `forloop.parentloop` 中。通常我们不用担心这个问题，但是一旦我们在模板中定义了 `forloop` 这个变量（当然我们反对这样做），在 `{% for %}` 块中它会在 `forloop.parentloop` 被重新命名。

ifequal/ifnotequal

Django模板系统压根儿就没想过实现一个全功能的编程语言，所以它不允许我们在模板中执行Python的语句（还是那句话，要了解更多请参看理念和限制小节）。 但是比较两个变量的值并且显示一些结果实在是个太常见的需求了，所以Django提供了 `{% ifequal %}` 标签供我们使用。

`{% ifequal %}` 标签比较两个值，当他们相等时，显示在 `{% ifequal %}` 和 `{% endifequal %}` 之中所有的值。

下面的例子比较两个模板变量 `user` 和 `currentuser` ：

```
{% ifequal user currentuser %}
    <h1>Welcome!</h1>
{% endifequal %}
```

参数可以是硬编码的字符串，随便用单引号或者双引号引起来，所以下列代码都是正确的：

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% endifequal %}

{% ifequal section "community" %}
    <h1>Community</h1>
{% endifequal %}
```

和 `{% if %}` 类似， `{% ifequal %}` 支持可选的 `{% else%}` 标签：

```
{% ifequal section 'sitenews' %}
    <h1>Site News</h1>
{% else %}
    <h1>No News Here</h1>
{% endifequal %}
```

只有模板变量，字符串，整数和小数可以作为 `{% ifequal %}` 标签的参数。下面是合法参数的例子：

```
{% ifequal variable 1 %}
{% ifequal variable 1.23 %}
{% ifequal variable 'foo' %}
{% ifequal variable "foo" %}
```

其他任何类型，例如Python的字典类型、列表类型、布尔类型，不能用在 `{% ifequal %}` 中。 下面是些错误的例子：

```
{% ifequal variable True %}
{% ifequal variable [1, 2, 3] %}
{% ifequal variable {'key': 'value'} %}
```

如果你需要判断变量是真还是假，请使用 `{% if %}` 来替代 `{% ifequal %}` 。

注释

就像HTML或者Python，Django模板语言同样提供代码注释。 注释使用 `{# #}` ：

```
{# This is a comment #}
```

注释的内容不会在模板渲染时输出。

用这种语法的注释不能跨越多行。 这个限制是为了提高模板解析的性能。 在下面这个模板中，输出结果和模板本身是 完全一样的（也就是说，注释标签并没有被解析为注释）：

```
This is a {# this is not
a comment #}
test.
```

如果要实现多行注释，可以使用 `{% comment %}` 模板标签，就像这样：

```
{% comment %}
This is a
multi-line comment.
{% endcomment %}
```

过滤器

就象本章前面提到的一样，模板过滤器是在变量被显示前修改它的值的一个简单方法。 过滤器使用管道字符，如下所示：

```
{{ name|lower }}
```

显示的内容是变量 `{{ name }}` 被过滤器 `lower` 处理后的结果，它功能是转换文本为小写。

过滤管道可以被 套接 ，既是说，一个过滤器管道的输出又可以作为下一个管道的输入，如此下去。 下面的例子实现查找列表的第一个元素并将其转化为大写。

```
{{ my_list|first|upper }}
```

有些过滤器有参数。 过滤器的参数跟随冒号之后并且总是以双引号包含。 例如：

```
{{ bio|truncatewords:"30" }}
```

这个将显示变量 `bio` 的前30个词。

以下几个是最为重要的过滤器的一部分。 附录F包含其余的过滤器。

> `addslashes`：添加反斜杠到任何反斜杠、单引号或者双引号前面。 这在处理包含JavaScript的文本时是非常有用的。
>
> `date`：按指定的格式字符串参数格式化 `date` 或者 `datetime` 对象， 范例：

```
{{ pub_date|date:"F j, Y" }}
```

> 格式参数的定义在附录F中。
>
> `length`：返回变量的长度。 对于列表，这个参数将返回列表元素的个数。 对于字符串，这个参数将返回字符串中字符的个数。 你可以对列表或者字符串，或者任何知道怎么测定长度的Python 对象使用这个方法（也就是说，有 `__len__()` 方法的对象）。

理念与局限

现在你已经对Django的模板语言有一些认识了，我们将指出一些特意设置的限制和为什么要这样做 背后的一些设计哲学。

相对与其他的网络应用的组件，模板的语法很具主观性，因此可供程序员的选择方案也很广泛。 事实上，Python有成十上百的 开放源码的模板语言实现。 每个实现都是因为开发者认为现存的模板语言不够用。 （事实上，对一个 Python开发者来说，写一个自己的模板语言就象是某种“成人礼”一样！ 如果你还没有完成一个自己的 模板语言，好好考虑写一个，这是一个非常有趣的锻炼。 ）

明白了这个，你也许有兴趣知道事实上Django并不强制要求你必须使用它的模板语言。 因为Django 虽然被设计成一个FULL-Stack的Web框架，它提供了开发者所必需的所有组件，而且在大多数情况 使用Django模板系统会比其他的Python模板库要 更方便 一点，但是并不是严格要求你必须使用 它。 你将在后续的“视图中应用模板”这一章节中看到，你还可以非常容易地在Django中使用其他的模板语言。

虽然如此，很明显，我们对Django模板语言的工作方式有着强烈的偏爱。 这个模板语言来源于World Online的开发经验和Django创造者们集体智慧的结晶。 下面是关于它的一些设计哲学理念：

> 业务逻辑应该和表现逻辑相对分开 。我们将模板系统视为控制表现及表现相关逻辑的工具，仅此而已。 模板系统不应提供超出此基本目标的功能。
>
> 出于这个原因，在 Django 模板中是不可能直接调用 Python 代码的。 所有的编程工作基本上都被局限于模板标签的能力范围。 当然， 是 有可能写出自定义的模板标签来完成任意工作，但这些“超范围”的 Django 模板标签有意地不允许执行任何 Python 代码。
>
> 语法不应受到 *HTML/XML* 的束缚 。尽管 Django 模板系统主要用于生成 HTML，它还是被有意地设计为可生成非 HTML 格式，如纯文本。 一些其它的模板语言是基于 XML 的，将所有的模板逻辑置于 XML 标签与属性之中，而 Django 有意地避开了这种限制。 强制要求使用有效 XML 编写模板将会引发大量的人为错误和难以理解的错误信息，而且使用 XML 引擎解析模板也会导致令人无法容忍的模板处理开销。
>
> 假定设计师精通 *HTML* 编码 。模板系统的设计意图并不是为了让模板一定能够很好地显示在 Dreamweaver 这样的所见即所得编辑器中。 这种限制过于苛刻，而且会使得语法不能像目前这样的完美。 Django 要求模板创作人员对直接编辑 HTML 非常熟悉。
>
> 假定设计师不是 *Python* 程序员 。模板系统开发人员认为：模板通常由设计师而非程序员来编写，因此不应被假定拥有Python开发知识。
>
> 当然，系统同样也特意地提供了对那些 由 Python 程序员进行模板制作的小型团队的支持。 它提供了一种工作模式，允许通过编写原生 Python 代码进行系统语法拓展。 （详见第十章）
>
> 目标并不是要发明一种编程语言 。目标是恰到好处地提供如分支和循环这一类编程式功能，这是进行与表现相关判断的基础。

在视图中使用模板

在学习了模板系统的基础之后，现在让我们使用相关知识来创建视图。 重新打开我们在前一章在 `mysite.views` 中创建的 `current_datetime` 视图。 以下是其内容：

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

让我们用 Django 模板系统来修改该视图。 第一步，你可能已经想到了要做下面这样的修改：

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t = Template("<html><body>It is now {{ current_date }}.</body></html>")
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

没错，它确实使用了模板系统，但是并没有解决我们在本章开头所指出的问题。 也就是说，模板仍然嵌入在Python代码里，并未真正的实现数据与表现的分离。 让我们将模板置于一个单独的文件 中，并且让视图加载该文件来解决此问题。

你可能首先考虑把模板保存在文件系统的某个位置并用 Python 内建的文件操作函数来读取文件内容。 假设文件保存在 `/home/djangouser/templates/mytemplate.html` 中的话，代码就会像下面这样:

```
from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    # Simple way of using templates from the filesystem.
    # This is BAD because it doesn't account for missing files!
    fp = open('/home/djangouser/templates/mytemplate.html')
    t = Template(fp.read())
    fp.close()
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

然而，基于以下几个原因，该方法还算不上简洁：

- 它没有对文件丢失的情况做出处理。 如果文件 `mytemplate.html` 不存在或者不可读， `open()` 函数调用将会引发 `IOError` 异常。
- 这里对模板文件的位置进行了硬编码。 如果你在每个视图函数都用该技术，就要不断复制这些模板的位置。 更不用说还要带来大量的输入工作！
- 它包含了大量令人生厌的重复代码。 与其在每次加载模板时都调用 `open()` 、 `fp.read()` 和 `fp.close()` ，还不如做出更佳选择。

为了解决这些问题，我们采用了 模板自加载 跟 模板目录 的技巧.

模板加载

为了减少模板加载调用过程及模板本身的冗余代码，Django 提供了一种使用方便且功能强大的 API ，用于从磁盘中加载模板，

要使用此模板加载API，首先你必须将模板的保存位置告诉框架。 设置的保存文件就是我们前一章节讲述 `ROOT_URLCONF` 配置的时候提到的 `settings.py` 。

如果你是一步步跟随我们学习过来的，马上打开你的 `settings.py` 配置文件，找到 `TEMPLATE_DIRS` 这项设置吧。 它的默认设置是一个空元组（tuple），加上一些自动生成的注释。

```
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/django/templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)
```

该设置告诉 Django 的模板加载机制在哪里查找模板。 选择一个目录用于存放模板并将其添加到 `TEMPLATE_DIRS` 中：

```
TEMPLATE_DIRS = (
    '/home/django/mysite/templates',
)
```

下面是一些注意事项：

> 你可以任意指定想要的目录，只要运行 Web 服务器的用户可以读取该目录的子目录和模板文件。 如果实在想不出合适的位置来放置模板，我们建议在 Django 项目中创建一个 `templates` 目录（也就是说，如果你一直都按本书的范例操作的话，在第二章创建的 `mysite` 目录中）。
>
> 如果你的 `TEMPLATE_DIRS` 只包含一个目录，别忘了在该目录后加上个逗号。
>
> Bad:

```
# Missing comma!
TEMPLATE_DIRS = (
    '/home/django/mysite/templates'
)
```

> Good:

```
# Comma correctly in place.
TEMPLATE_DIRS = (
    '/home/django/mysite/templates',
)
```

> Python 要求单元素元组中必须使用逗号，以此消除与圆括号表达式之间的歧义。 这是新手常犯的错误。
>
> 如果使用的是 Windows 平台，请包含驱动器符号并使用Unix风格的斜杠（/）而不是反斜杠（），就像下面这样：

```
TEMPLATE_DIRS = (
    'C:/www/django/templates',
)
```

> 最省事的方式是使用绝对路径（即从文件系统根目录开始的目录路径）。 如果想要更灵活一点并减少一些负面干扰，可利用 Django 配置文件就是 Python 代码这一点来动态构建 `TEMPLATE_DIRS` 的内容，如： 例如：

```
import os.path

TEMPLATE_DIRS = (
    os.path.join(os.path.dirname(__file__), 'templates').replace('\\','/'),
)
```

> 这个例子使用了神奇的 Python 内部变量 `__file__` ，该变量被自动设置为代码所在的 Python 模块文件名。 `os.path.dirname(__file__)` 将会获取自身所在的文件，即`settings.py` 所在的目录，然后由`os.path.join` 这个方法将这目录与 `templates` 进行连接。如果在windows下，它会智能地选择正确的后向斜杠”“进行连接，而不是前向斜杠”/”。
>
> 在这里我们面对的是动态语言python代码，我需要提醒你的是，不要在你的设置文件里写入错误的代码，这很重要。 如果你在这里引入了语法错误，或运行错误，你的Django-powered站点将很可能就要被崩溃掉。

完成 `TEMPLATE_DIRS` 设置后，下一步就是修改视图代码，让它使用 Django 模板加载功能而不是对模板路径硬编码。 返回 `current_datetime` 视图，进行如下修改：

```
from django.template.loader import get_template
from django.template import Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t = get_template('current_datetime.html')
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

此范例中，我们使用了函数 `django.template.loader.get_template()` ，而不是手动从文件系统加载模板。 该 `get_template()` 函数以模板名称为参数，在文件系统中找出模块的位置，打开文件并返回一个编译好的 `Template` 对象。

在这个例子里，我们选择的模板文件是 `current_datetime.html` ，但这个与 `.html` 后缀没有直接的联系。 你可以选择任意后缀的任意文件，只要是符合逻辑的都行。甚至选择没有后缀的文件也不会有问题。

要确定某个模板文件在你的系统里的位置， `get_template()` 方法会自动为你连接已经设置的 `TEMPLATE_DIRS` 目录和你传入该法的模板名称参数。比如，你的 `TEMPLATE_DIRS` 目录设置为 `'/home/django/mysite/templates'` ，上面的 `get_template()` 调用就会为你找到 `/home/django/mysite/templates/current_datetime.html` 这样一个位置。

如果 `get_template()` 找不到给定名称的模板，将会引发一个 `TemplateDoesNotExist` 异常。要了解究竟会发生什么，让我们按照第三章内容，在 Django 项目目录中运行 `python manage.py runserver` 命令，再次启动Django开发服务器。 接着，告诉你的浏览器，使其定位到指定页面以激活 `current_datetime` 视图（如 `http://127.0.0.1:8000/time/` ）。假设你的 `DEBUG` 项设置为 `True` ，而你有没有建立 `current_datetime.html` 这个模板文件，你会看到Django的错误提示网页，告诉你发生了 `TemplateDoesNotExist` 错误。

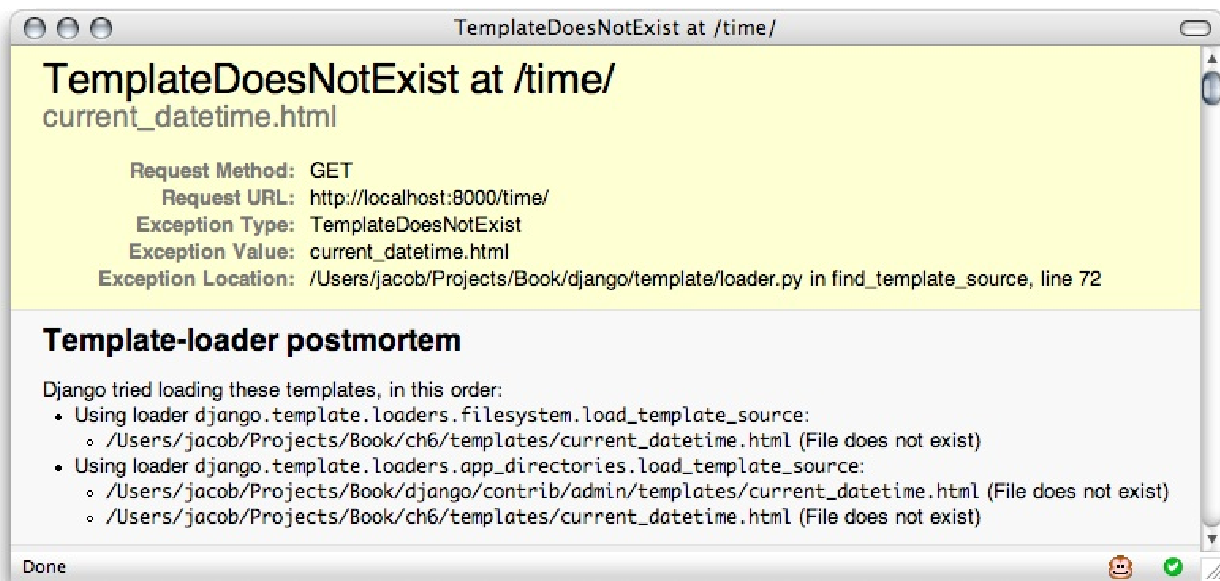


图 4-1: 模板文件无法找到时，将会发送提示错误的网页给用户。

该页面与我们在第三章解释过的错误页面相似，只不过多了一块调试信息区： 模板加载器事后检查区。 该区域显示 Django 要加载哪个模板、每次尝试出错的原因（如：文件不存在等）。 当你尝试调试模板加载错误时，这些信息会非常有帮助。

接下来，在模板目录中创建包括以下模板代码 `current_datetime.html` 文件：

```
<html><body>It is now {{ current_date }}.</body></html>
```

在网页浏览器中刷新该页，你将会看到完整解析后的页面。

render_to_response()

我们已经告诉你如何载入一个模板文件，然后用 `Context` 渲染它，最后返回这个处理好的 `HttpResponse` 对象给用户。 我们已经优化了方案，使用 `get_template()` 方法代替繁杂的用代码来处理模板及其路径的工作。 但这仍然需要一定量的时间来敲出这些简化的代码。 这是一个普遍存在的重复苦力劳动。Django为此提供了一个捷径，让你一次性地载入某个模板文件，渲染它，然后将此作为 `HttpResponse` 返回。

该捷径就是位于 `django.shortcuts` 模块中名为 `render_to_response()` 的函数。大多数情况下，你会使用```\ ``[``](#id21)``对象，除非你的老板以代码行数来衡量你的工作。

System Message: WARNING/2 (`<string>` , line 1736); *backlink*

Inline literal start-string without end-string.

System Message: WARNING/2 (`<string>` , line 1736); *backlink*

Inline literal start-string without end-string.

System Message: WARNING/2 (`<string>` , line 1736); *backlink*

Inline literal start-string without end-string.

下面就是使用 `render_to_response()` 重新编写过的 `current_datetime` 范例。

```
from django.shortcuts import render_to_response
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html', {'current_date': now})
```

大变样了！ 让我们逐句看看代码发生的变化：

- 我们不再需要导入 `get_template` 、 `Template` 、 `Context` 和 `HttpResponse` 。相反，我们导入 `django.shortcuts.render_to_response` 。 `import datetime` 继续保留.
- 在 `current_datetime` 函数中，我们仍然进行 `now` 计算，但模板加载、上下文创建、模板解析和 `HttpResponse` 创建工作均在对 `render_to_response()` 的调用中完成了。 由于 `render_to_response()` 返回 `HttpResponse` 对象，因此我们仅需在视图中 `return` 该值。

`render_to_response()` 的第一个参数必须是要使用的模板名称。 如果要给定第二个参数，那么该参数必须是为该模板创建 `Context` 时所使用的字典。 如果不提供第二个参数， `render_to_response()` 使用一个空字典。

locals() 技巧

思考一下我们对 `current_datetime` 的最后一次赋值:

```
def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html', {'current_date': now})
```

很多时候，就像在这个范例中那样，你发现自己一直在计算某个变量，保存结果到变量中（比如前面代码中的 now ），然后将这些变量发送给模板。 尤其喜欢偷懒的程序员应该注意到了，不断地为临时变量和临时模板命名有那么一点点多余。 不仅多余，而且需要额外的输入。

如果你是个喜欢偷懒的程序员并想让代码看起来更加简明，可以利用 Python 的内建函数 `locals()` 。它返回的字典对所有局部变量的名称与值进行映射。 因此，前面的视图可以重写成下面这个样子：

```
def current_datetime(request):
    current_date = datetime.datetime.now()
    return render_to_response('current_datetime.html', locals())
```

在此，我们没有像之前那样手工指定 context 字典，而是传入了 `locals()` 的值，它囊括了函数执行到该时间点时所定义的一切变量。 因此，我们将 `now` 变量重命名为 `current_date` ，因为那才是模板所预期的变量名称。 在本例中， `locals()` 并没有带来多 大 的改进，但是如果有多个模板变量要界定而你又想偷懒，这种技术可以减少一些键盘输入。

使用 `locals()` 时要注意是它将包括 所有 的局部变量，它们可能比你想让模板访问的要多。 在前例中， `locals()` 还包含了 `request` 。对此如何取舍取决你的应用程序。

get_template()中使用子目录

把所有的模板都存放在一个目录下可能会让事情变得难以掌控。 你可能会考虑把模板存放在你模板目录的子目录中，这非常好。 事实上，我们推荐这样做；一些Django的高级特性（例如将在第十一章讲到的通用视图系统）的缺省约定就是期望使用这种模板布局。

把模板存放于模板目录的子目录中是件很轻松的事情。 只需在调用 `get_template()` 时，把子目录名和一条斜杠添加到模板名称之前，如：

```
t = get_template('dateapp/current_datetime.html')
```

由于 `render_to_response()` 只是对 `get_template()` 的简单封装， 你可以对 `render_to_response()` 的第一个参数做相同处理。

```
return render_to_response('dateapp/current_datetime.html', {'current_date': now})
```

对子目录树的深度没有限制，你想要多少层都可以。 只要你喜欢，用多少层的子目录都无所谓。

注意

Windows用户必须使用斜杠而不是反斜杠。 `get_template()` 假定的是 Unix 风格的文件名符号约定。

`include` 模板标签

在讲解了模板加载机制之后，我们再介绍一个利用该机制的内建模板标签： `{% include %}` 。该标签允许在（模板中）包含其它的模板的内容。 标签的参数是所要包含的模板名称，可以是一个变量，也可以是用单/双引号硬编码的字符串。 每当在多个模板中出现相同的代码时，就应该考虑是否要使用 `{% include %}` 来减少重复。

下面这两个例子都包含了 `nav.html` 模板。这两个例子是等价的，它们证明单/双引号都是允许的。

```
{% include 'nav.html' %}
{% include "nav.html" %}
```

下面的例子包含了 `includes/nav.html` 模板的内容:

```
{% include 'includes/nav.html' %}
```

下面的例子包含了以变量 `template_name` 的值为名称的模板内容：

```
{% include template_name %}
```

和在 `get_template()` 中一样， 对模板的文件名进行判断时会在所调取的模板名称之前加上来自 `TEMPLATE_DIRS` 的模板目录。

所包含的模板执行时的 context 和包含它们的模板是一样的。 举例说，考虑下面两个模板文件：

```
# mypage.html

<html>
<body>
{% include "includes/nav.html" %}
<h1>{{ title }}</h1>
</body>
</html>

# includes/nav.html

<div id="nav">
    You are in: {{ current_section }}
</div>
```

如果你用一个包含 `current_section` 的上下文去渲染 `mypage.html` 这个模板文件，这个变量将存在于它所包含（include）的模板里，就像你想象的那样。

如果 `{% include %}` 标签指定的模板没找到，Django将会在下面两个处理方法中选择一个：

- 如果 `DEBUG` 设置为 `True` ，你将会在 Django 错误信息页面看到 `TemplateDoesNotExist` 异常。
- 如果 `DEBUG` 设置为 `False` ，该标签不会引发错误信息，在标签位置不显示任何东西。

模板继承

到目前为止，我们的模板范例都只是些零星的 HTML 片段，但在实际应用中，你将用 Django 模板系统来创建整个 HTML 页面。 这就带来一个常见的 Web 开发问题： 在整个网站中，如何减少共用页面区域（比如站点导航）所引起的重复和冗余代码？

解决该问题的传统做法是使用 服务器端的 *includes* ，你可以在 HTML 页面中使用该指令将一个网页嵌入到另一个中。 事实上， Django 通过刚才讲述的 `{% include %}` 支持了这种方法。 但是用 Django 解决此类问题的首选方法是使用更加优雅的策略—— 模板继承 。

本质上来说，模板继承就是先构造一个基础框架模板，而后在其子模板中对它所包含站点公用部分和定义块进行重载。

让我们通过修改 `current_datetime.html` 文件，为 `current_datetime` 创建一个更加完整的模板来体会一下这种做法：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>The current time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>It is now {{ current_date }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

这看起来很棒，但如果我们要为第三章的 `hours_ahead` 视图创建另一个模板会发生什么事情呢？

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Future time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

很明显，我们刚才重复了大量的 HTML 代码。 想象一下，如果有一个更典型的网站，它有导航条、样式表，可能还有一些 JavaScript 代码，事情必将以向每个模板填充各种冗余的 HTML 而告终。

解决这个问题的服务器端 include 方案是找出两个模板中的共同部分，将其保存为不同的模板片段，然后在每个模板中进行 include。 也许你会把模板头部的一些代码保存为 `header.html` 文件：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
```

你可能会把底部保存到文件 `footer.html` ：

```
    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

对基于 include 的策略，头部和底部的包含很简单。 麻烦的是中间部分。 在此范例中，每个页面都有一个 `<h1>My helpful timestamp site</h1>` 标题，但是这个标题不能放在 `header.html` 中，因为每个页面的 `<title>` 是不同的。 如果我们将 `<h1>` 包含在头部，我们就不得不包含 `<title>` ，但这样又不允许在每个页面对它进行定制。 何去何从呢？

Django 的模板继承系统解决了这些问题。 你可以将其视为服务器端 include 的逆向思维版本。 你可以对那些 不同 的代码段进行定义，而不是 共同 代码段。

第一步是定义 基础模板 ， 该框架之后将由 子模板 所继承。 以下是我们目前所讲述范例的基础模板：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    {% block content %}{% endblock %}
    {% block footer %}
    <hr>
    <p>Thanks for visiting my site.</p>
    {% endblock %}
</body>
</html>
```

这个叫做 `base.html` 的模板定义了一个简单的 HTML 框架文档，我们将在本站点的所有页面中使用。 子模板的作用就是重载、添加或保留那些块的内容。 （如果你一直按顺序学习到这里，保存这个文件到你的template目录下，命名为 `base.html` .）

我们使用一个以前已经见过的模板标签： `{% block %}` 。 所有的 `{% block %}` 标签告诉模板引擎，子模板可以重载这些部分。 每个 `{% block %}` 标签所要做的是告诉模板引擎，该模板下的这一块内容将有可能被子模板覆盖。

现在我们已经有了一个基本模板，我们可以修改 `current_datetime.html` 模板来 使用它：

```
{% extends "base.html" %}

{% block title %}The current time{% endblock %}

{% block content %}
<p>It is now {{ current_date }}.</p>
{% endblock %}
```

再为 `hours_ahead` 视图创建一个模板，看起来是这样的：

```
{% extends "base.html" %}

{% block title %}Future time{% endblock %}

{% block content %}
<p>In {{ hour_offset }} hour(s), it will be {{ next_time }}.</p>
{% endblock %}
```

看起来很漂亮是不是？ 每个模板只包含对自己而言 独一无二 的代码。 无需多余的部分。 如果想进行站点级的设计修改，仅需修改 `base.html` ，所有其它模板会立即反映出所作修改。

以下是其工作方式。 在加载 `current_datetime.html` 模板时，模板引擎发现了 `{% extends %}` 标签， 注意到该模板是一个子模板。 模板引擎立即装载其父模板，即本例中的 `base.html` 。

此时，模板引擎注意到 `base.html` 中的三个 `{% block %}` 标签，并用子模板的内容替换这些 block 。因此，引擎将会使用我们在 `{ block title %}` 中定义的标题，对 `{% block content %}` 也是如此。 所以，网页标题一块将由 `{% block title %}` 替换，同样

地，网页的内容一块将由 `{% block content %}` 替换。

注意由于子模板并没有定义 `footer` 块，模板系统将使用在父模板中定义的值。 父模板 `{% block %}` 标签中的内容总是被当作一条退路。

继承并不会影响到模板的上下文。 换句话说，任何处在继承树上的模板都可以访问到你传到模板中的每一个模板变量。

你可以根据需要使用任意多的继承次数。 使用继承的一种常见方式是下面的三层法：

1. 创建 `base.html` 模板，在其中定义站点的主要外观感受。 这些都是不常修改甚至从不修改的部分。

2. 为网站的每个区域创建 `base_SECTION.html` 模板(例如, `base_photos.html` 和 `base_forum.html`)。这些模板对 `base.html` 进行拓展，并包含区域特定的风格与设计。

3. 为每种类型的页面创建独立的模板，例如论坛页面或者图片库。 这些模板拓展相应的区域模板。

这个方法可最大限度地重用代码，并使得向公共区域（如区域级的导航）添加内容成为一件轻松的工作。

以下是使用模板继承的一些诀窍：

- 如果在模板中使用 `{% extends %}` ，必须保证其为模板中的第一个模板标记。 否则，模板继承将不起作用。

- 一般来说，基础模板中的 `{% block %}` 标签越多越好。 记住，子模板不必定义父模板中所有的代码块，因此你可以用合理的缺省值对一些代码块进行填充，然后只对子模板所需的代码块进行（重）定义。 俗话说，钩子越多越好。

- 如果发觉自己在多个模板之间拷贝代码，你应该考虑将该代码段放置到父模板的某个 `{% block %}` 中。

- 如果你需要访问父模板中的块的内容，使用 `{{ block.super }}`这个标签吧，这一个魔法变量将会表现出父模板中的内容。 如果只想在上级代码块基础上添加内容，而不是全部重载，该变量就显得非常有用了。

- 不允许在同一个模板中定义多个同名的 `{% block %}` 。 存在这样的限制是因为block 标签的工作方式是双向的。 也就是说，block 标签不仅挖了一个要填的坑，也定义了在父模板中这个坑所填充的内容。如果模板中出现了两个相同名称的 *{% block %}* 标签，父模板将无从得知要使用哪个块的内容。

- `{% extends %}` 对所传入模板名称使用的加载方法和 `get_template()` 相同。 也就是说，会将模板名称被添加到 `TEMPLATE_DIRS` 设置之后。

- 多数情况下， `{% extends %}` 的参数应该是字符串，但是如果直到运行时方能确定父模板名，这个参数也可以是个变量。 这使得你能够实现一些很酷的动态功能。

下一章

你现在已经掌握了模板系统的基本知识。 接下来呢？

时下大多数网站都是 数据库驱动 的：网站的内容都是存储在关系型数据库中。 这使得数据和逻辑能够彻底地分开（视图和模板也以同样方式对逻辑和显示进行了分隔。）

下一章将讲述如何与数据库打交道。

第五章：模型

在第三章，我们讲述了用 Django 建造网站的基本途径： 建立视图和 URLConf 。 正如我们所阐述的，视图负责处理一些主观逻辑，然后返回响应结果。 作为例子之一，我们的主观逻辑是要计算当前的日期和时间。

在当代 Web 应用中，主观逻辑经常牵涉到与数据库的交互。 数据库驱动网站 在后台连接数据库服务器，从中取出一些数据，然后在 Web 页面用漂亮的格式展示这些数据。 这个网站也可能会向访问者提供修改数据库数据的方法。

许多复杂的网站都提供了以上两个功能的某种结合。 例如 Amazon.com 就是一个数据库驱动站点的良好范例。 本质上，每个产品页面都是数据库中数据以 HTML格式进行的展现，而当你发表客户评论时，该评论被插入评论数据库中。

由于先天具备 Python 简单而强大的数据库查询执行方法，Django 非常适合开发数据库驱动网站。 本章深入介绍了该功能： Django 数据库层。

（注意： 尽管对 Django 数据库层的使用中并不特别强调这点，但是我们还是强烈建议您掌握一些数据库和 SQL 原理。 对这些概念的介绍超越了本书的范围，但就算你是数据库方面的菜鸟，我们也建议你继续阅读。 你也许能够跟上进度，并在上下文学习过程中掌握一些概念。）

在视图中进行数据库查询的笨方法

正如第三章详细介绍的那个在视图中输出 HTML 的笨方法（通过在视图里对文本直接硬编码HTML），在视图中也有笨方法可以从数据库中获取数据。 很简单： 用现有的任何 Python 类库执行一条 SQL 查询并对结果进行一些处理。

在本例的视图中，我们使用了 `MySQLdb` 类库（可以从 http://www.djangoproject.com/r/python-mysql/ 获得）来连接 MySQL 数据库，取回一些记录，将它们提供给模板以显示一个网页：

```
from django.shortcuts import render_to_response
import MySQLdb

def book_list(request):
    db = MySQLdb.connect(user='me', db='mydb', passwd='secret', host='localhost')
    cursor = db.cursor()
    cursor.execute('SELECT name FROM books ORDER BY name')
    names = [row[0] for row in cursor.fetchall()]
    db.close()
    return render_to_response('book_list.html', {'names': names})
```

这个方法可用，但很快一些问题将出现在你面前：

- 我们将数据库连接参数硬行编码于代码之中。 理想情况下，这些参数应当保存在 Django 配置中。
- 我们不得不重复同样的代码： 创建数据库连接、创建数据库游标、执行某个语句、然后关闭数据库。 理想情况下，我们所需要应该只是指定所需的结果。
- 它把我们栓死在 MySQL 之上。 如果过段时间，我们要从 MySQL 换到 PostgreSQL，就不得不使用不同的数据库适配器（例如 `psycopg` 而不是 `MySQLdb` ），改变连接参数，根据 SQL 语句的类型可能还要修改SQL 。 理想情况下，应对所使用的数据库服务器进行抽象，这样一来只在一处修改即可变换数据库服务器。 （如果你正在建立一个开源的Django应用程序来尽可能让更多人使用的话，这个特性是非常适当的。）

正如你所期待的，Django数据库层正是致力于解决这些问题。 以下提前揭示了如何使用 Django 数据库 API 重写之前那个视图。

```
from django.shortcuts import render_to_response
from mysite.books.models import Book

def book_list(request):
    books = Book.objects.order_by('name')
    return render_to_response('book_list.html', {'books': books})
```

我们将在本章稍后的地方解释这段代码。 目前而言，仅需对它有个大致的认识。

MTV 开发模式

在钻研更多代码之前，让我们先花点时间考虑下 Django 数据驱动 Web 应用的总体设计。

我们在前面章节提到过，Django 的设计鼓励松耦合及对应用程序中不同部分的严格分割。 遵循这个理念的话，要想修改应用的某部分而不影响其它部分就比较容易了。 在视图函数中，我们已经讨论了通过模板系统把业务逻辑和表现逻辑分隔开的重要性。 在数据库层中，我们对数据访问逻辑也应用了同样的理念。

把数据存取逻辑、业务逻辑和表现逻辑组合在一起的概念有时被称为软件架构的 *Model-View-Controller* (MVC)模式。 在这个模式中， Model 代表数据存取层，View 代表的是系统中选择显示什么和怎么显示的部分，Controller 指的是系统中根据用户输入并视需要访问模型，以决定使用哪个视图的那部分。

为什么用缩写？

像 MVC 这样的明确定义模式的主要用于改善开发人员之间的沟通。 比起告诉同事，“让我们采用抽象的数据存取方式，然后单独划分一层来显示数据，并且在中间加上一个控制它的层”，一个通用的说法会让你收益，你只需要说：“我们在这里使用MVC模式吧。”。

Django 紧紧地遵循这种 MVC 模式，可以称得上是一种 MVC 框架。 以下是 Django 中 M、V 和 C 各自的含义：

- *M*，数据存取部分，由django数据库层处理，本章要讲述的内容。
- *V*，选择显示哪些数据要显示以及怎样显示的部分，由视图和模板处理。
- *C*，根据用户输入委派视图的部分，由 Django 框架根据 URLconf 设置，对给定 URL 调用适当的 Python 函数。

由于 C 由框架自行处理，而 Django 里更关注的是模型（Model）、模板(Template)和视图（Views），Django 也被称为 *MTV* 框架 。在 MTV 开发模式中：

- *M* 代表模型（Model），即数据存取层。 该层处理与数据相关的所有事务： 如何存取、如何验证有效性、包含哪些行为以及数据之间的关系等。
- *T* 代表模板(Template)，即表现层。 该层处理与表现相关的决定： 如何在页面或其他类型文档中进行显示。
- *V* 代表视图（View），即业务逻辑层。 该层包含存取模型及调取恰当模板的相关逻辑。 你可以把它看作模型与模板之间的桥梁。

如果你熟悉其它的 MVC Web开发框架，比方说 Ruby on Rails，你可能会认为 Django 视图是控制器，而 Django 模板是视图。 很不幸，这是对 MVC 不同诠释所引起的错误认识。 在 Django 对 MVC 的诠释中，视图用来描述要展现给用户的数据；不是数据 如何展现 ,而且展现 哪些 数据。 相比之下，Ruby on Rails 及一些同类框架提倡控制器负责决定向用户展现哪些数据，而视图则仅决定 如何 展现数据，而不是展现 哪些 数据。

两种诠释中没有哪个更加正确一些。 重要的是要理解底层概念。

数据库配置

记住这些理念之后，让我们来开始 Django 数据库层的探索。 首先，我们需要做些初始配置；我们需要告诉Django使用什么数据库以及如何连接数据库。

我们假定你已经完成了数据库服务器的安装和激活，并且已经在其中创建了数据库（例如，用 `CREATE DATABASE` 语句）。 如果你使用SQLite，不需要这步安装，因为SQLite使用文件系统上的独立文件来存储数据。

象前面章节提到的 `TEMPLATE_DIRS` 一样，数据库配置也是在Django的配置文件里，缺省 是 `settings.py` 。 打开这个文件并查找数据库配置：

```
DATABASE_ENGINE = ''
DATABASE_NAME = ''
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_HOST = ''
DATABASE_PORT = ''
```

配置纲要如下。

`DATABASE_ENGINE` 告诉Django使用哪个数据库引擎。 如果你在 Django 中使用数据库， `DATABASE_ENGINE` 必须是 Table 5-1 中所列出的值。

<table class="cn docutils" id="cn43"><caption>表 5-1. 数据库引擎设置 </caption> <colgroup><col width="28%"> <col width="14%"> <col width="58%"></colgroup>

<thead valign="bottom">

<tr><th>设置</th><th>数据库</th><th>所需适配器</th></tr>

</thead>

<tbody valign="top">

<tr><td> `postgresql` </td><td>PostgreSQL</td><td> `psycopg` 1.x版， http://www.djangoproject.com/r/python-pgsql/1/。 </td></tr>

<tr><td> `postgresql_psycopg2` </td><td>PostgreSQL</td><td> `psycopg` 2.x版， http://www.djangoproject.com/r/python-pgsql/。 </td></tr>

<tr><td> `mysql` </td><td>MySQL</td><td> `MySQLdb` , http://www.djangoproject.com/r/python-mysql/.</td></tr>

<tr><td> `sqlite3` </td><td>SQLite</td><td>如果使用Python 2.5+则不需要适配器。 否则就使用 `pysqlite` ， http://www.djangoproject.com/r/python-sqlite/。</td></tr>

<tr><td> `oracle` </td><td>Oracle</td><td> `cx_Oracle` , http://www.djangoproject.com/r/python-oracle/.</td></tr>

</tbody>

</table>

要注意的是无论选择使用哪个数据库服务器，都必须下载和安装对应的数据库适配器。 访问表 5-1 中“所需适配器”一栏中的链接，可通过互联网免费获取这些适配器。 如果你使用Linux，你的发布包管理系统会提供合适的包。 比如说查找 `python-postgresql` 或者 `python-psycopg` 的软件包。

配置示例：

```
DATABASE_ENGINE = 'postgresql_psycopg2'
```

`DATABASE_NAME` 将数据库名称告知 Django 。 例如：

```
DATABASE_NAME = 'mydb'
```

如果使用 SQLite，请对数据库文件指定完整的文件系统路径。 例如：

```
DATABASE_NAME = '/home/django/mydata.db'
```

> 在这个例子中，我们将SQLite数据库放在/home/django目录下，你可以任意选用最合适你的目录。
>
> `DATABASE_USER` 告诉 Django 用哪个用户连接数据库。 例如： 如果用SQLite，空白即可。
>
> `DATABASE_PASSWORD` 告诉Django连接用户的密码。 SQLite 用空密码即可。
>
> `DATABASE_HOST` 告诉 Django 连接哪一台主机的数据库服务器。 如果数据库与 Django 安装于同一台计算机（即本机），可将此项保留空白。 如果你使用SQLite，此项留空。
>
> 此处的 MySQL 是一个特例。 如果使用的是 MySQL 且该项设置值由斜杠（ `'/'` ）开头，MySQL 将通过 Unix socket 来连接指定的套接字，例如：

```
DATABASE_HOST = '/var/run/mysql'
```

一旦在输入了那些设置并保存之后应当测试一下你的配置。 我们可以在 `mysite` 项目目录下执行上章所提到的 `python manage.py shell` 来进行测试。 （我们上一章提到过在， `manager.py shell` 命令是以正确Django配置启用Python交互解释器的一种方法。 这个方法在这里是很有必要的，因为Django需要知道加载哪个配置文件来获取数据库连接信息。）

输入下面这些命令来测试你的数据库配置：

```
>>> from django.db import connection
>>> cursor = connection.cursor()
```

如果没有显示什么错误信息，那么你的数据库配置是正确的。 否则，你就得 查看错误信息来纠正错误。 表 5-2 是一些常见错误。

表 5-2\. 数据库配置错误信息

错误信息	解决方法
You haven't set the DATABASE_ENGINE setting yet.	不要以空字符串配置`` DATABASE_ENGINE`` 的值。 表格 5-1 列出可用的值。
Environment variable DJANGO_SETTINGS_MODULE is undefined.	使用`` python manager.py shell`` 命令启动交互解释器，不要以`` python`` 命令直接启动交互解释器。
Error loading _____ module: No module named _____.	未安装合适的数据库适配器 (例如, `psycopg` 或 `MySQLdb`)。Django并不自带适配器，所以你得自己下载安装。
_____ isn't an available database backend.	把`DATABASE_ENGINE` 配置成前面提到的合法的数据库引擎。 也许是拼写错误？
database _____ does not exist	设置`` DATABASE_NAME`` 指向存在的数据库，或者先在数据库客户端中执行合适的`` CREATE DATABASE`` 语句创建数据库。
role _____ does not exist	设置`` DATABASE_USER`` 指向存在的用户，或者先在数据库客户端中执创建用户。
could not connect to server	查看DATABASE_HOST和DATABASE_PORT是否已正确配置，并确认数据库服务器是否已正常运行。

第一个应用程序

你现在已经确认数据库连接正常工作了，让我们来创建一个 *Django app*-一个包含模型，视图和Django代码，并且形式为独立Python包的完整Django应用。

在这里要先解释一些术语，初学者可能会混淆它们。 在第二章我们已经创建了 *project* , 那么 *project* 和 *app* 之间到底有什么不同呢？它们的区别就是一个是配置另一个是 代码：

> 一个project包含很多个Django app以及对它们的配置。
>
> 技术上，project的作用是提供配置文件，比方说哪里定义数据库连接信息, 安装的app列表， `TEMPLATE_DIRS` ， 等等。
>
> 一个app是一套Django功能的集合，通常包括模型和视图，按Python的包结构的方式存在。
>
> 例如，Django本身内建有一些app，例如注释系统和自动管理界面。 app的一个关键点是它们是很容易移植到其他project和被多个project复用。

对于如何架构Django代码并没有快速成套的规则。 如果你只是建造一个简单的Web站点，那么可能你只需要一个app就可以了； 但如果是一个包含许多不相关的模块的复杂的网站，例如电子商务和社区之类的站点，那么你可能需要把这些模块划分成不同的app，以便以后复用。

不错，你可以不用创建app，这一点应经被我们之前编写的视图函数的例子证明了 。 在那些例子中，我们只是简单的创建了一个称为 `views.py` 的文件，编写了一些函数并在URLconf中设置了各个函数的映射。 这些情况都不需要使用apps。

但是，系统对app有一个约定： 如果你使用了Django的数据库层（模型），你 必须创建一个Django app。 模型必须存放在apps中。 因此，为了开始建造 我们的模型，我们必须创建一个新的app。

在 `mysite` 项目文件下输入下面的命令来创建 `books` app：

```
python manage.py startapp books
```

这个命令并没有输出什么，它只在 `mysite` 的目录里创建了一个 `books` 目录。 让我们来看看这个目录的内容：

```
books/
    __init__.py
    models.py
    tests.py
    views.py
```

这个目录包含了这个app的模型和视图。

使用你最喜欢的文本编辑器查看一下 `models.py` 和 `views.py` 文件的内容。 它们都是空的，除了 `models.py` 里有一个 import。这就是你Django app的基础。

在Python代码里定义模型

我们早些时候谈到。MTV里的M代表模型。 Django模型是用Python代码形式表述的数据在数据库中的定义。 对数据层来说它等同于 CREATE TABLE 语句，只不过执行的是Python代码而不是 SQL，而且还包含了比数据库字段定义更多的含义。 Django用模型在后台执行SQL代码并把结果用Python的数据结构来描述。 Django也使用模型来呈现SQL无法处理的高级概念。

如果你对数据库很熟悉，你可能马上就会想到，用Python 和 SQL来定义数据模型是不是有点多余？ Django这样做是有下面几个原因的：

> 自省（运行时自动识别数据库）会导致过载和有数据完整性问题。 为了提供方便的数据访问API， Django需要以 某种方式 知道数据库层内部信息，有两种实现方式。 第一种方式是用Python明确地定义数据模型，第二种方式是通过自省来自动侦测识别数据模型。
>
> 第二种方式看起来更清晰，因为数据表信息只存放在一个地方-数据库里，但是会带来一些问题。 首先，运行时扫描数据库会带来严重的系统过载。 如果每个请求都要扫描数据库的表结构，或者即便是 服务启动时做一次都是会带来不能接受的系统过载。 （有人认为这个程度的系统过载是可以接受的，而Django开发者的目标是尽可能地降低框架的系统过载）。第二，某些数据库，尤其是老版本的MySQL,并未完整存储那些精确的自省元数据。
>
> 编写Python代码是非常有趣的，保持用Python的方式思考会避免你的大脑在不同领域来回切换。 尽可能的保持在单一的编程环境/思想状态下可以帮助你提高生产率。 不得不去重复写SQL，再写Python代码，再写SQL，…，会让你头都要裂了。
>
> 把数据模型用代码的方式表述来让你可以容易对它们进行版本控制。 这样，你可以很容易了解数据层 的变动情况。
>
> SQL只能描述特定类型的数据字段。 例如，大多数数据库都没有专用的字段类型来描述Email地址、URL。 而用Django的模型可以做到这一点。 好处就是高级的数据类型带来更高的效率和更好的代码复用。
>
> SQL还有在不同数据库平台的兼容性问题。 发布Web应用的时候，使用Python模块描述数据库结构信息可以避免为MySQL, PostgreSQL, and SQLite编写不同的 `CREATE TABLE` 。

当然，这个方法也有一个缺点，就是Python代码和数据库表的同步问题。 如果你修改了一个Django模型， 你要自己来修改数据库来保证和模型同步。 我们将在稍后讲解解决这个问题的几种策略。

最后,我们要提醒你Django提供了实用工具来从现有的数据库表中自动扫描生成模型。 这对已有的数据库来说是非常快捷有用的。 我们将在第18章中对此进行讨论。

第一个模型

在本章和后续章节里，我们把注意力放在一个基本的 书籍/作者/出版商 数据库结构上。 我们这样做是因为 这是一个众所周知的例子，很多SQL有关的书籍也常用这个举例。 你现在看的这本书也是由作者 创作再由出版商出版的哦！

我们来假定下面的这些概念、字段和关系：

- 一个作者有姓，有名及email地址。
- 出版商有名称，地址，所在城市、省，国家，网站。

- 书籍有书名和出版日期。 它有一个或多个作者（和作者是多对多的关联关系[many-to-many]）， 只有一个出版商（和出版商是一对多的关联关系[one-to-many]，也被称作外键[foreign key]）

第一步是用Python代码来描述它们。 打开由 `startapp` 命令创建的 `models.py` 并输入下面的内容：

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

让我们来快速讲解一下这些代码的含义。 首先要注意的事是每个数据模型都是 `django.db.models.Model` 的子类。它的父类 Model 包含了所有必要的和数据库交互的方法，并提供了一个简洁漂亮的定义数据库字段的语法。 信不信由你，这些就是我们需要编写的通过Django存取基本数据的所有代码。

每个模型相当于单个数据库表，每个属性也是这个表中的一个字段。 属性名就是字段名，它的类型（例如 `CharField` ）相当于数据库的字段类型 （例如 `varchar` ）。例如， `Publisher` 模块等同于下面这张表（用PostgreSQL的 `CREATE TABLE` 语法描述）：

```
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
);
```

事实上，正如过一会儿我们所要展示的，Django 可以自动生成这些 `CREATE TABLE` 语句。

“每个数据库表对应一个类”这条规则的例外情况是多对多关系。 在我们的范例模型中， `Book` 有一个 `多对多字段` 叫做 `authors` 。 该字段表明一本书籍有一个或多个作者，但 `Book` 数据库表却并没有 `authors` 字段。 相反，Django创建了一个额外的表（多对多连接表）来处理书籍和作者之间的映射关系。

请查看附录 B 了解所有的字段类型和模型语法选项。

最后需要注意的是，我们并没有显式地为这些模型定义任何主键。 除非你单独指明，否则Django会自动为每个模型生成一个自增长的整数主键字段每个Django模型都要求有单独的主键。 `id`

模型安装

完成这些代码之后，现在让我们来在数据库中创建这些表。 要完成该项工作，第一步是在 Django 项目中 激活 这些模型。 将 `books` app 添加到配置文件的已安装应用列表中即可完成此步骤。

再次编辑 `settings.py` 文件， 找到 `INSTALLED_APPS` 设置。 `INSTALLED_APPS` 告诉 Django 项目哪些 app 处于激活状态。 缺省情况下如下所示：

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
)
```

把这四个设置前面加#临时注释起来。 （这四个app是经常使用到的，我们将在后续章节里讨论如何使用它们）。同时，注释掉MIDDLEWARE_CLASSES的默认设置条目，因为这些条目是依赖于刚才我们刚在INSTALLED_APPS注释掉的apps。 然后，添加 `‘mysite.books’` 到 `INSTALLED_APPS` 的末尾，此时设置的内容看起来应该是这样的：

```
MIDDLEWARE_CLASSES = (
    # 'django.middleware.common.CommonMiddleware',
    # 'django.contrib.sessions.middleware.SessionMiddleware',
    # 'django.contrib.auth.middleware.AuthenticationMiddleware',
)

INSTALLED_APPS = (
    # 'django.contrib.auth',
    # 'django.contrib.contenttypes',
    # 'django.contrib.sessions',
    # 'django.contrib.sites',
    'mysite.books',
)
```

(就像我们在上一章设置TEMPLATE*DIRS*所提到的逗号，同样在*INSTALLED_APPS*的末尾也需添加一个逗号，因为这是个单元素的元组。 另外，本书的作者喜欢在 每一个_ tuple元素后面加一个逗号，不管它是不是 只有一个元素。 这是为了避免忘了加逗号，而且也没什么坏处。)

`'mysite.books'` 指示我们正在编写的 `books` app。 `INSTALLED_APPS` 中的每个app都使用Python的路径描述，包的路径，用小数点“.”间隔。

现在我们可以创建数据库表了。 首先，用下面的命令验证模型的有效性：

```
python manage.py validate
```

`validate` 命令检查你的模型的语法和逻辑是否正确。 如果一切正常，你会看到 `0 errors found` 消息。如果出错，请检查你输入的模型代码。 错误输出会给出非常有用的错误信息来帮助你修正你的模型。

一旦你觉得你的模型可能有问题，运行 `python manage.py validate` 。 它可以帮助你捕获一些常见的模型定义错误。

模型确认没问题了，运行下面的命令来生成 `CREATE TABLE` 语句（如果你使用的是Unix，那么可以启用语法高亮）：

```
python manage.py sqlall books
```

在这个命令行中， `books` 是app的名称。 和你运行 `manage.py startapp` 中的一样。执行之后，输出如下：

```
BEGIN;
CREATE TABLE "books_publisher" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(30) NOT NULL,
    "address" varchar(50) NOT NULL,
    "city" varchar(60) NOT NULL,
    "state_province" varchar(30) NOT NULL,
    "country" varchar(50) NOT NULL,
    "website" varchar(200) NOT NULL
)
;
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL
)
;
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id") DEFERRABLE INITIA
    "publication_date" date NOT NULL
)
;
CREATE TABLE "books_book_authors" (
    "id" serial NOT NULL PRIMARY KEY,
    "book_id" integer NOT NULL REFERENCES "books_book" ("id") DEFERRABLE INITIALLY DEFERR
    "author_id" integer NOT NULL REFERENCES "books_author" ("id") DEFERRABLE INITIALLY DE
    UNIQUE ("book_id", "author_id")
)
;
CREATE INDEX "books_book_publisher_id" ON "books_book" ("publisher_id");
COMMIT;
```

注意：

- 自动生成的表名是app名称（ `books` ）和模型的小写名称 （ `publisher` ，`book` ，`author`)的组合。你可以参考附录B重写这个规则。
- 我们前面已经提到，Django为每个表格自动添加加了一个 `id` 主键， 你可以重新设置它。
- 按约定，Django添加 `"_id"` 后缀到外键字段名。 你猜对了，这个同样是可以自定义的。
- 外键是用 `REFERENCES` 语句明确定义的。
- 这些 `CREATE TABLE` 语句会根据你的数据库而作调整，这样象数据库特定的一些字段例如：（MySQL）， `auto_increment` （PostgreSQL）, `serial` （SQLite），都会自动生成。 `integer primary key` 同样的，字段名称也是自动处理（例如单引号还好是双引号）。 例子中的输出是基于PostgreSQL语法的。

`sqlall` 命令并没有在数据库中真正创建数据表，只是把SQL语句段打印出来，这样你可以看到Django究竟会做些什么。 如果你想这么做的话，你可以把那些SQL语句复制到你的数据库客户端执行，或者通过Unix管道直接进行操作（例如， `python manager.py sqlall books | psql mydb` ）。不过，Django提供了一种更为简易的提交SQL语句至数据库的方法： `syncdb` 命令

```
python manage.py syncdb
```

执行这个命令后，将看到类似以下的内容：

```
Creating table books_publisher
Creating table books_author
Creating table books_book
Installing index for books.Book model
```

`syncdb` 命令是同步你的模型到数据库的一个简单方法。 它会根据 `INSTALLED_APPS` 里设置的app来检查数据库， 如果表不存在，它就会创建它。 需要注意的是， `syncdb` 并 不能将模型的修改或删除同步到数据库；如果你修改或删除了一个模型，并想把它提交到数据库， `syncdb` 并不会做出任何处理。 （更多内容请查看本章最后的“修改数据库的架构”一段。）

如果你再次运行 `python manage.py syncdb` ，什么也没发生，因为你没有添加新的模型或者添加新的app。因此，运行 `python manage.py syncdb` 总是安全的，因为它不会重复执行SQL语句。

如果你有兴趣，花点时间用你的SQL客户端登录进数据库服务器看看刚才Django创建的数据表。 你可以手动启动命令行客户端（例如，执行PostgreSQL的 `psql` 命令），也可以执行 `python manage.py dbshell` ，这个命令将依据 `DATABASE_SERVER` 的里设置自动检测使用哪种命令行客户端。 常言说，后来者居上。

基本数据访问

一旦你创建了模型，Django自动为这些模型提供了高级的Python API。 运行 `python manage.py shell` 并输入下面的内容试试看：

```
>>> from books.models import Publisher
>>> p1 = Publisher(name='Apress', address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p1.save()
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',
...     city='Cambridge', state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> p2.save()
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

这短短几行代码干了不少的事。 这里简单的说一下：

- 首先，导入Publisher模型类， 通过这个类我们可以与包含 出版社 的数据表进行交互。
- 接着，创建一个 `Publisher` 类的实例并设置了字段 `name, address` 等的值。
- 调用该对象的 `save()` 方法，将对象保存到数据库中。 Django 会在后台执行一条 `INSERT` 语句。
- 最后，使用 `Publisher.objects` 属性从数据库取出出版商的信息，这个属性可以认为是包含出版商的记录集。 这个属性有许多方法， 这里先介绍调用 `Publisher.objects.all()` 方法获取数据库中 `Publisher` 类的所有对象。这个操作的幕后，Django执行了一条SQL `SELECT` 语句。

这里有一个值得注意的地方，在这个例子可能并未清晰地展示。 当你使用Django modle API创建对象时Django并未将对象保存至数据库内，除非你调用 `save()` 方法：

```
p1 = Publisher(...)
# At this point, p1 is not saved to the database yet!
p1.save()
# Now it is.
```

如果需要一步完成对象的创建与存储至数据库，就使用 `objects.create()` 方法。 下面的例子与之前的例子等价：

```
>>> p1 = Publisher.objects.create(name='Apress',
...     address='2855 Telegraph Avenue',
...     city='Berkeley', state_province='CA', country='U.S.A.',
...     website='http://www.apress.com/')
>>> p2 = Publisher.objects.create(name="O'Reilly",
...     address='10 Fawcett St.', city='Cambridge',
...     state_province='MA', country='U.S.A.',
...     website='http://www.oreilly.com/')
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
```

当然，你肯定想执行更多的Django数据库API试试看，不过，还是让我们先解决一点烦人的小问题。

添加模块的字符串表现

当我们打印整个publisher列表时，我们没有得到想要的有用信息，无法把````对象区分开来：

System Message: WARNING/2 (`<string>` , line 872); *backlink*

Inline literal start-string without end-string.

System Message: WARNING/2 (`<string>` , line 872); *backlink*

Inline literal start-string without end-string.

```
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

我们可以简单解决这个问题，只需要为 `Publisher` 对象添加一个方法 `__unicode__()` 。 `__unicode__()` 方法告诉Python如何将对象以unicode的方式显示出来。 为以上三个模型添加 `__unicode__()` 方法后，就可以看到效果了：

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    **def __unicode__(self):**
        **return self.name**

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

    **def __unicode__(self):**
        **return u'%s %s' % (self.first_name, self.last_name)**

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    **def __unicode__(self):**
        **return self.title**
```

就象你看到的一样， `__unicode__()` 方法可以进行任何处理来返回对一个对象的字符串表示。 `Publisher` 和 `Book` 对象的 `__unicode__()` 方法简单地返回各自的名称和标题， `Author` 对象的 `__unicode__()` 方法则稍微复杂一些，它将 `first_name` 和 `last_name` 字段

值以空格连接后再返回。

对**unicode**()的唯一要求就是它要返回一个unicode对象 如果 `__unicode__()` 方法未返回一个Unicode对象，而返回比如说一个整型数字，那么Python将抛出一个 `TypeError` 错误，并提示："coercing to Unicode: need string or buffer, int found" 。

Unicode对象

什么是Unicode对象呢？

你可以认为unicode对象就是一个Python字符串，它可以处理上百万不同类别的字符——从古老版本的Latin字符到非Latin字符，再到曲折的引用和艰涩的符号。

普通的python字符串是经过编码的，意思就是它们使用了某种编码方式（如ASCII，ISO-8859-1或者UTF-8）来编码。 如果你把奇特的字符（其它任何超出标准128个如0-9和A-Z之类的ASCII字符）保存在一个普通的Python字符串里，你一定要跟踪你的字符串是用什么编码的，否则这些奇特的字符可能会在显示或者打印的时候出现乱码。 当你尝试要将用某种编码保存的数据结合到另外一种编码的数据中，或者你想要把它显示在已经假定了某种编码的程序中的时候，问题就会发生。 我们都已经见到过网页和邮件被???弄得乱七八糟。 ?????? 或者其它出现在奇怪位置的字符：这一般来说就是存在编码问题了。

但是Unicode对象并没有编码。它们使用Unicode，一个一致的，通用的字符编码集。 当你在Python中处理Unicode对象的时候，你可以直接将它们混合使用和互相匹配而不必去考虑编码细节。

Django 在其内部的各个方面都使用到了 Unicode 对象。 模型 对象中，检索匹配方面的操作使用的是 Unicode 对象，视图 函数之间的交互使用的是 Unicode 对象，模板的渲染也是用的 Unicode 对象。 通常，我们不必担心编码是否正确，后台会处理的很好。

注意，我们这里只是对Unicode对象进行非常浅显的概述，若要深入了解你可能需要查阅相关的资料。 这是一个很好的起点：http://www.joelonsoftware.com/articles/Unicode.html。

为了让我们的修改生效，先退出Python Shell，然后再次运行 `python manage.py shell` 进入。（这是保证代码修改生效的最简单方法。）现在 `Publisher` 对象列表容易理解多了。

```
>>> from books.models import Publisher
>>> publisher_list = Publisher.objects.all()
>>> publisher_list
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

请确保你的每一个模型里都包含 `__unicode__()` 方法，这不只是为了交互时方便，也是因为Django会在其他一些地方用 `__unicode__()` 来显示对象。

最后， `__unicode__()` 也是一个很好的例子来演示我们怎么添加 行为 到模型里。 Django的模型不只是为对象定义了数据库表的结构，还定义了对象的行为。 `__unicode__()` 就是一个例子来演示模型知道怎么显示它们自己。

插入和更新数据

你已经知道怎么做了： 先使用一些关键参数创建对象实例，如下：

```
>>> p = Publisher(name='Apress',
...         address='2855 Telegraph Ave.',
...         city='Berkeley',
...         state_province='CA',
...         country='U.S.A.',
...         website='http://www.apress.com/')
```

这个对象实例并 没有 对数据库做修改。 在调用 `save()` 方法之前，记录并没有保存至数据库，像这样：

```
>>> p.save()
```

在SQL里，这大致可以转换成这样：

```
INSERT INTO books_publisher
    (name, address, city, state_province, country, website)
VALUES
    ('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA',
     'U.S.A.', 'http://www.apress.com/');
```

因为 `Publisher` 模型有一个自动增加的主键 `id` ，所以第一次调用 `save()` 还多做了一件事： 计算这个主键的值并把它赋值给这个对象实例：

```
>>> p.id
52    # this will differ based on your own data
```

接下来再调用 `save()` 将不会创建新的记录，而只是修改记录内容（也就是 执行 `UPDATE` SQL语句，而不是 `INSERT` 语句）：

```
>>> p.name = 'Apress Publishing'
>>> p.save()
```

前面执行的 `save()` 相当于下面的SQL语句：

```
UPDATE books_publisher SET
    name = 'Apress Publishing',
    address = '2855 Telegraph Ave.',
    city = 'Berkeley',
    state_province = 'CA',
    country = 'U.S.A.',
    website = 'http://www.apress.com'
WHERE id = 52;
```

注意，并不是只更新修改过的那个字段，所有的字段都会被更新。 这个操作有可能引起竞态条件，这取决于你的应用程序。 请参阅后面的“更新多个对象”小节以了解如何实现这种轻量的修改（只修改对象的部分字段）。

```
UPDATE books_publisher SET
    name = 'Apress Publishing'
WHERE id=52;
```

选择对象

当然，创建新的数据库，并更新之中的数据是必要的，但是，对于 Web 应用程序来说，更多的时候是在检索查询数据库。 我们已经知道如何从一个给定的模型中取出所有记录：

```
>>> Publisher.objects.all()
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

这相当于这个SQL语句：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher;
```

注意

注意到Django在选择所有数据时并没有使用 `SELECT*` ，而是显式列出了所有字段。 设计的时候就是这样： `SELECT*` 会更慢，而且最重要的是列出所有字段遵循了Python 界的一个信条： 明言胜于暗示。

有关Python之禅(戒律) :-），在Python提示行输入 `import this` 试试看。

让我们来仔细看看 `Publisher.objects.all()` 这行的每个部分：

> 首先，我们有一个已定义的模型 `Publisher` 。没什么好奇怪的： 你想要查找数据， 你就用模型来获得数据。
>
> 然后，是 `objects` 属性。 它被称为管理器，我们将在第10章中详细讨论它。 目前，我们只需了解管理器管理着所有针对数据包含、还有最重要的数据查询的表格级操作。
>
> 所有的模型都自动拥有一个 `objects` 管理器；你可以在想要查找数据时使用它。
>
> 最后，还有 `all()` 方法。这个方法返回返回数据库中所有的记录。 尽管这个对象 看起来 象一个列表（list），它实际是一个 QuerySet 对象， 这个对象是数据库中一些记录的集合。 附录C将详细描述QuerySet。 现在，我们就先当它是一个仿真列表对象好了。

所有的数据库查找都遵循一个通用模式：

数据过滤

我们很少会一次性从数据库中取出所有的数据；通常都只针对一部分数据进行操作。 在Django API中，我们可以使用 `filter()` 方法对数据进行过滤：

```
>>> Publisher.objects.filter(name='Apress')
[<Publisher: Apress>]
```

`filter()` 根据关键字参数来转换成 `WHERE` SQL语句。 前面这个例子 相当于这样：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE name = 'Apress';
```

你可以传递多个参数到 `filter()` 来缩小选取范围：

```
>>> Publisher.objects.filter(country="U.S.A.", state_province="CA")
[<Publisher: Apress>]
```

多个参数会被转换成 `AND` SQL从句， 因此上面的代码可以转化成这样：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE country = 'U.S.A.'
AND state_province = 'CA';
```

注意，SQL缺省的 `=` 操作符是精确匹配的， 其他类型的查找也可以使用：

```
>>> Publisher.objects.filter(name__contains="press")
[<Publisher: Apress>]
```

在 *name* 和 `contains` 之间有双下划线。和Python一样，Django也使用双下划线来表明会进行一些魔术般的操作。这里， `contains` 部分会被Django翻译成 `LIKE` 语句：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE name LIKE '%press%';
```

其他的一些查找类型有： `icontains` (大小写无关的 `LIKE`), `startswith` 和 `endswith` , 还有 `range` (SQL `BETWEEN` 查询）。 附录C详细描述了所有的查找类型。

获取单个对象

上面的例子中 `filter()` 函数返回一个记录集，这个记录集是一个列表。 相对列表来说，有些时候我们更需要获取单个的对象， `get()` 方法就是在此时使用的：

```
>>> Publisher.objects.get(name="Apress")
<Publisher: Apress>
```

这样，就返回了单个对象，而不是列表（更准确的说，QuerySet)。 所以，如果结果是多个对象，会导致抛出异常：

```
>>> Publisher.objects.get(country="U.S.A.")
Traceback (most recent call last):
    ...
MultipleObjectsReturned: get() returned more than one Publisher --
    it returned 2! Lookup parameters were {'country': 'U.S.A.'}
```

如果查询没有返回结果也会抛出异常：

```
>>> Publisher.objects.get(name="Penguin")
Traceback (most recent call last):
    ...
DoesNotExist: Publisher matching query does not exist.
```

这个 `DoesNotExist` 异常 是 Publisher 这个 model 类的一个属性，即 `Publisher.DoesNotExist` 。在你的应用中，你可以捕获并处理这个异常，像这样：

```
try:
    p = Publisher.objects.get(name='Apress')
except Publisher.DoesNotExist:
    print "Apress isn't in the database yet."
else:
    print "Apress is in the database."
```

数据排序

在运行前面的例子中，你可能已经注意到返回的结果是无序的。 我们还没有告诉数据库 怎样对结果进行排序，所以我们返回的结果是无序的。

在你的 Django 应用中，你或许希望根据某字段的值对检索结果排序，比如说，按字母顺序。 那么，使用 `order_by()` 这个方法就可以搞定了。

```
>>> Publisher.objects.order_by("name")
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

跟以前的 `all()` 例子差不多，SQL语句里多了指定排序的部分：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name;
```

我们可以对任意字段进行排序：

```
>>> Publisher.objects.order_by("address")
[<Publisher: O'Reilly>, <Publisher: Apress>]

>>> Publisher.objects.order_by("state_province")
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

如果需要以多个字段为标准进行排序（第二个字段会在第一个字段的值相同的情况下被使用到），使用多个参数就可以了，如下：

```
>>> Publisher.objects.order_by("state_province", "address")
 [<Publisher: Apress>, <Publisher: O'Reilly>]
```

我们还可以指定逆向排序，在前面加一个减号 `-` 前缀：

```
>>> Publisher.objects.order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

尽管很灵活，但是每次都要用 `order_by()` 显得有点啰嗦。 大多数时间你通常只会对某些 字段进行排序。 在这种情况下，Django让你可以指定模型的缺省排序方式：

```
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __unicode__(self):
        return self.name

    **class Meta:**
        **ordering = ['name']**
```

现在，让我们来接触一个新的概念。 `class Meta`，内嵌于 `Publisher` 这个类的定义中（如果 `class Publisher` 是顶格的，那么 `class Meta` 在它之下要缩进4个空格－－按 Python 的传统 ）。你可以在任意一个 模型 类中使用 `Meta` 类，来设置一些与特定模型相关的选项。 在 附录B 中有 `Meta` 中所有可选项的完整参考，现在，我们关注 `ordering` 这个选项就够了。 如果你设置了这个选项，那么除非你检索时特意额外地使用了 `order_by()`，否则，当你使用 Django 的数据库 API 去检索时，`Publisher` 对象的相关返回值默认地都会按 `name` 字段排序。

连锁查询

我们已经知道如何对数据进行过滤和排序。 当然，通常我们需要同时进行过滤和排序查询的操作。 因此，你可以简单地写成这种“链式”的形式：

```
>>> Publisher.objects.filter(country="U.S.A.").order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

你应该没猜错，转换成SQL查询就是 `WHERE` 和 `ORDER BY` 的组合：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE country = 'U.S.A'
ORDER BY name DESC;
```

限制返回的数据

另一个常用的需求就是取出固定数目的记录。 想象一下你有成千上万的出版商在你的数据库里， 但是你只想显示第一个。 你可以使用标准的Python列表裁剪语句：

```
>>> Publisher.objects.order_by('name')[0]
<Publisher: Apress>
```

这相当于：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
LIMIT 1;
```

类似的，你可以用Python的range-slicing语法来取出数据的特定子集：

```
>>> Publisher.objects.order_by('name')[0:2]
```

这个例子返回两个对象，等同于以下的SQL语句：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
OFFSET 0 LIMIT 2;
```

注意，不支持Python的负索引(negative slicing)：

```
>>> Publisher.objects.order_by('name')[-1]
Traceback (most recent call last):
  ...
AssertionError: Negative indexing is not supported.
```

虽然不支持负索引，但是我们可以使用其他的方法。 比如，稍微修改 order_by() 语句来实现：

```
>>> Publisher.objects.order_by('-name')[0]
```

更新多个对象

在"插入和更新数据"小节中，我们有提到模型的save()方法，这个方法会更新一行里的所有列。 而某些情况下，我们只需要更新行里的某几列。

例如说我们现在想要将Apress Publisher的名称由原来的"Apress"更改为"Apress Publishing"。若使用save()方法，如：

```
>>> p = Publisher.objects.get(name='Apress')
>>> p.name = 'Apress Publishing'
>>> p.save()
```

这等同于如下SQL语句：

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE name = 'Apress';

UPDATE books_publisher SET
    name = 'Apress Publishing',
    address = '2855 Telegraph Ave.',
    city = 'Berkeley',
    state_province = 'CA',
    country = 'U.S.A.',
    website = 'http://www.apress.com'
WHERE id = 52;
```

（注意在这里我们假设Apress的ID为52）

在这个例子里我们可以看到Django的save()方法更新了不仅仅是name列的值，还有更新了所有的列。 若name以外的列有可能会被其他的进程所改动的情况下，只更改name列显然是更加明智的。 更改某一指定的列，我们可以调用结果集（QuerySet）对象的update()方法： 示例如下：

```
>>> Publisher.objects.filter(id=52).update(name='Apress Publishing')
```

与之等同的SQL语句变得更高效，并且不会引起竞态条件。

```
UPDATE books_publisher
SET name = 'Apress Publishing'
WHERE id = 52;
```

update()方法对于任何结果集（QuerySet）均有效，这意味着你可以同时更新多条记录。 以下示例演示如何将所有Publisher的country字段值由'U.S.A'更改为'USA'：

```
>>> Publisher.objects.all().update(country='USA')
2
```

update()方法会返回一个整型数值，表示受影响的记录条数。 在上面的例子中，这个值是2。

删除对象

删除数据库中的对象只需调用该对象的delete()方法即可：

```
>>> p = Publisher.objects.get(name="O'Reilly")
>>> p.delete()
>>> Publisher.objects.all()
[<Publisher: Apress Publishing>]
```

同样我们可以在结果集上调用delete()方法同时删除多条记录。这一点与我们上一小节提到的update()方法相似：

```
>>> Publisher.objects.filter(country='USA').delete()
>>> Publisher.objects.all().delete()
>>> Publisher.objects.all()
[]
```

删除数据时要谨慎！ 为了预防误删除掉某一个表内的所有数据，Django要求在删除表内所有数据时显示使用all()。 比如，下面的操作将会出错：

```
>>> Publisher.objects.delete()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: 'Manager' object has no attribute 'delete'
```

而一旦使用all()方法，所有数据将会被删除：

```
>>> Publisher.objects.all().delete()
```

如果只需要删除部分的数据，就不需要调用all()方法。再看一下之前的例子：

```
>>> Publisher.objects.filter(country='USA').delete()
```

下一章

通过本章的学习，你应该可以熟练地使用Django模型来编写一些简单的数据库应用程序。 在第十章我们将讨论Django数据库层的高级应用。

一旦你定义了你的模型，接下来就是要把数据导入数据库里了。 你可能已经有现成的数据了，请看第十八章以获得有关如何集成现有数据库的建议。 也可能数据是用户提供的，第七章中还会教你怎么处理用户提交的数据。

有时候，你和你的团队成员也需要手工输入数据，这时候如果有一个基于Web的数据输入和管理的界面就会很有帮助。 下一章将介绍解决手工录入问题的方法——Django管理界面。

第六章：Django站点管理

对于某一类网站， 管理界面 是基础设施中非常重要的一部分。 这是以网页和有限的可信任管理者为基础的界面，它可以让你添加，编辑和删除网站内容。 一些常见的例子： 你可以用这个界面发布博客，后台的网站管理者用它来润色读者提交的内容，你的客户用你给他们建立的界面工具更新新闻并发布在网站上，这些都是使用管理界面的例子。

但是管理界面有一问题： 创建它太繁琐。 当你开发对公众的功能时，网页开发是有趣的，但是创建管理界面通常是千篇一律的。 你必须认证用户，显示并管理表格，验证输入的有效性诸如此类。 这很繁琐而且是重复劳动。

Django 在对这些繁琐和重复的工作进行了哪些改进？ 它用不能再少的代码为你做了所有的一切。 Django 中创建管理界面已经不是问题。

这一章是关于 Django 的自动管理界面。 这个特性是这样起作用的： 它读取你模式中的元数据，然后提供给你一个强大而且可以使用的界面，网站管理者可以用它立即工作。

请注意我们建议你读这章，即使你不打算用admin。因为我们将介绍一些概念，这些概念可以应用到Django的所有方面，而不仅仅是admin

django.contrib 包

Django自动管理工具是django.contrib的一部分。django.contrib是一套庞大的功能集，它是Django基本代码的组成部分，Django框架就是由众多包含附加组件(add-on)的基本代码构成的。 你可以把django.contrib看作是可选的Python标准库或普遍模式的实际实现。 它们与Django捆绑在一起，这样你在开发中就不用“重复发明轮子”了。

管理工具是本书讲述django.contrib的第一个部分。从技术层面上讲，它被称作django.contrib.admin。django.contrib中其它可用的特性，如用户鉴别系统(django.contrib.auth)、支持匿名会话(django.contrib.sessioins)以及用户评注系统(django.contrib.comments)。这些，我们将在第十六章详细讨论。在成为一个Django专家以前，你将会知道更多django.contrib的特性。 目前，你只需要知道Django自带很多优秀的附加组件，它们都存在于django.contrib包里。

激活管理界面

Django管理站点完全是可选择的，因为仅仅某些特殊类型的站点才需要这些功能。 这意味着你需要在你的项目中花费几个步骤去激活它。

第一步，对你的settings文件做如下这些改变：

1. 将 `'django.contrib.admin'` 加入setting的 `INSTALLED_APPS` 配置中 （ `INSTALLED_APPS` 中的配置顺序是没有关系的, 但是我们喜欢保持一定顺序以方便人来阅读）

2. 保证 `INSTALLED_APPS` 中包含 `'django.contrib.auth'` ， `'django.contrib.contenttypes'` 和 `'django.contrib.sessions'` ，Django的管理工具需要这3个包。 (如果你跟随本文制作mysite项目的话，那么请注意我们在第五章的时候把这三项INSTALLED_APPS条目注释了。现在，请把注释取消。)

3. 确保 `MIDDLEWARE_CLASSES` 包含 `'django.middleware.common.CommonMiddleware'` 、 `'django.contrib.sessions.middleware.SessionMiddleware'` 和 `'django.contrib.auth.middleware.AuthenticationMiddleware'` 。(再次提醒，如果有跟着做mysite的话，请把在第五章做的注释取消。)

运行 `python manage.py syncdb` 。这一步将生成管理界面使用的额外数据库表。 当你把 `'django.contrib.auth'` 加进 `INSTALLED_APPS` 后，第一次运行 `syncdb` 命令时, 系统会请你创建一个超级用户。 如果你不这么作，你需要运行 `python manage.py createsuperuser` 来另外创建一个admin的用户帐号，否则你将不能登入admin (提醒一句: 只有当 `INSTALLED_APPS` 包含 `'django.contrib.auth'` 时， `python manage.py createsuperuser` 这个命令才可用.)

第三，将admin访问配置在URLconf(记住，在 `urls.py` 中). 默认情况下，命令 `django-admin.py startproject` 生成的文件 `urls.py` 是将Django admin的路径注释掉的，你所要做的就是取消注释。 请注意，以下内容是必须确保存在的：

```
# Include these import statements...
from django.contrib import admin
admin.autodiscover()

# And include this URLpattern...
urlpatterns = patterns('',
    # ...
    (r'^admin/', include(admin.site.urls)),
    # ...
)
```

当这一切都配置好后，现在你将发现Django管理工具可以运行了。 启动开发服务器(如前： `python manage.py runserver`)，然后在浏览器中访问： `http://127.0.0.1:8000/admin/`

，使用管理工具。

管理界面的设计是针对非技术人员的，所以它应该是自我解释的。 尽管如此，这里简单介绍一下它的基本特性。

你看到的第一件事是如图6-1所示的登录屏幕。

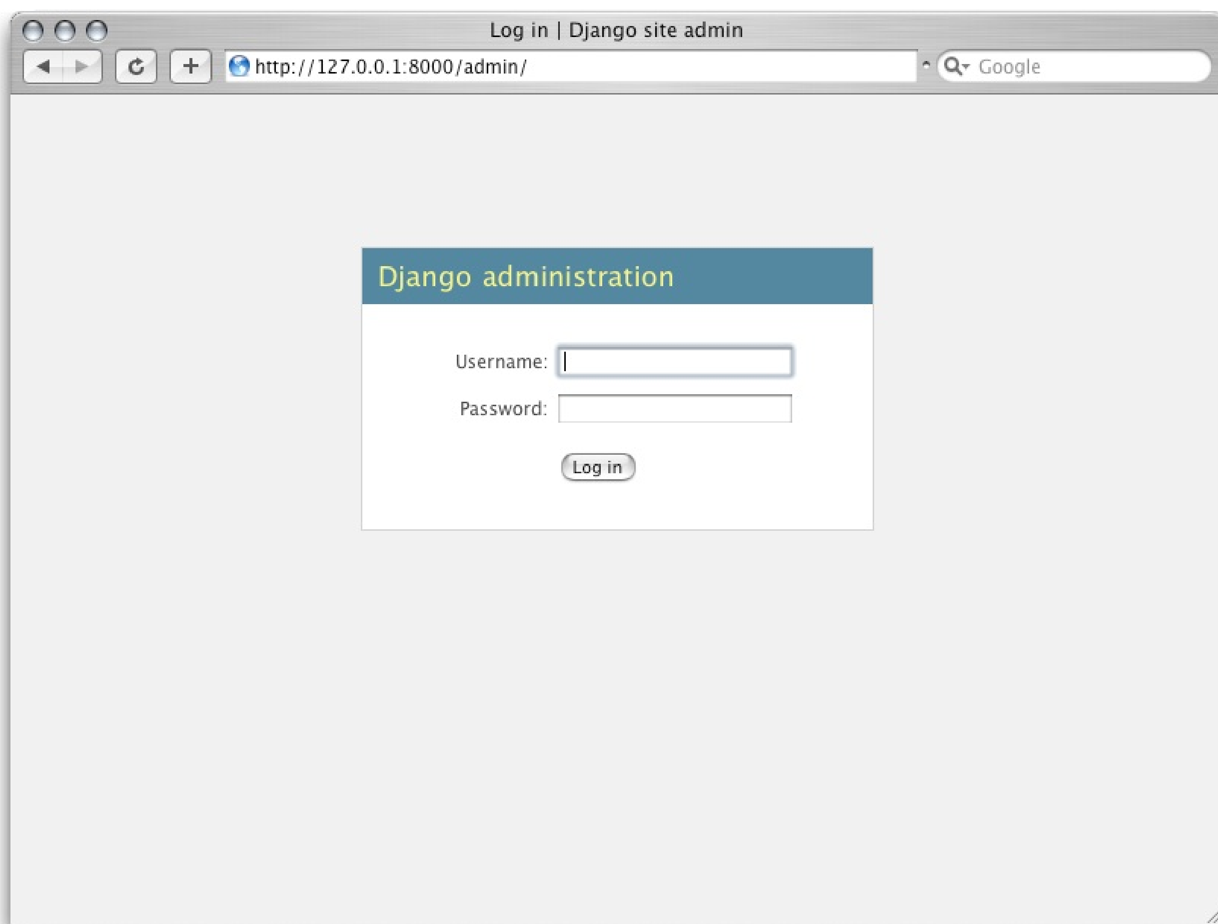


图 6-1. Django的登录截图

你要使用你原来设置的超级用户的用户名和密码。 如果无法登录，请运行 `python manage.py createsuperuser` ，确保你已经创建了一个超级用户。

一旦登录了，你将看到管理页面。 这个页面列出了管理工具中可编辑的所有数据类型。 现在，由于我们还没有创建任何模块，所以这个列表只有寥寥数条类目： 它仅有两个默认的管理-编辑模块：用户组(Groups)和用户(Users)。

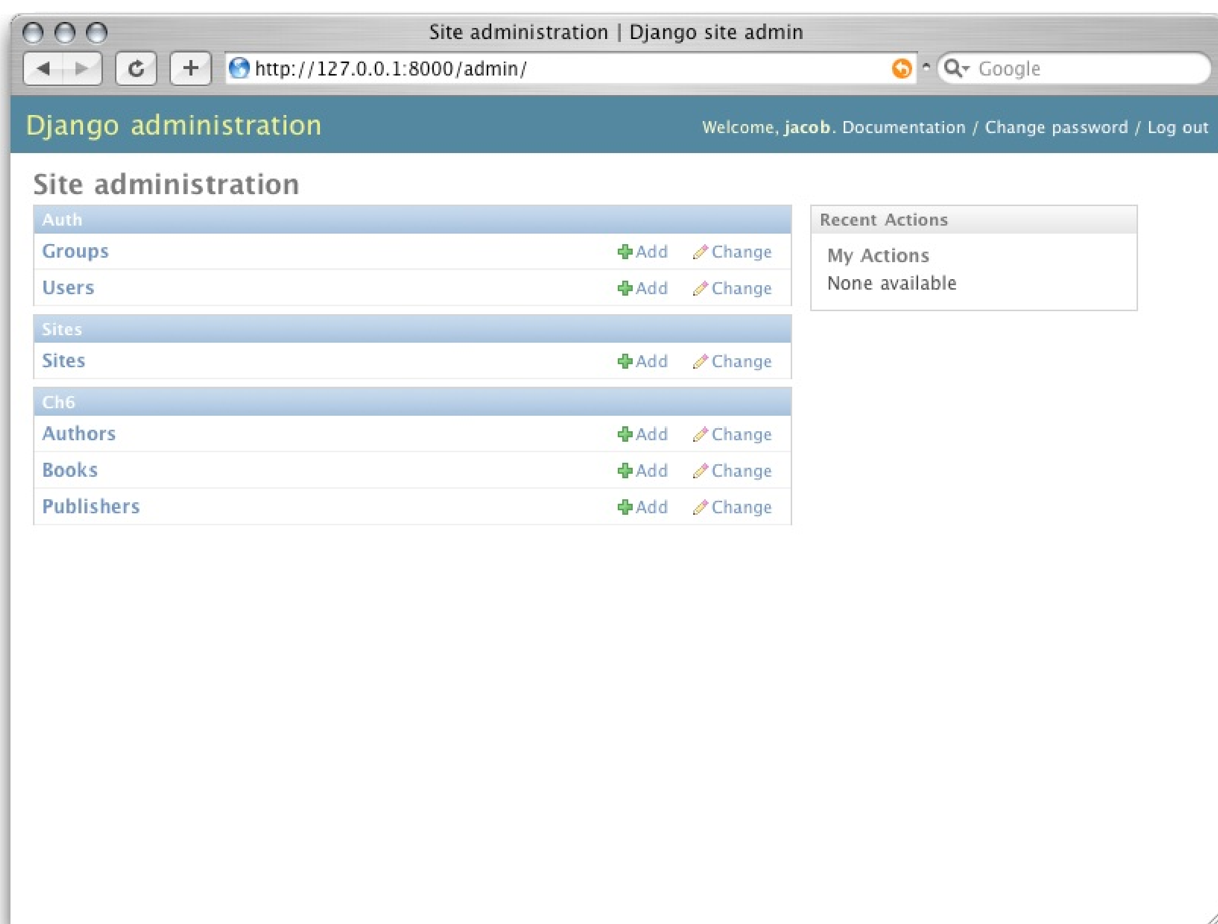


图 6-2。 Django admin的首页

在Django管理页面中，每一种数据类型都有一个 *change list* 和 *edit form* 。前者显示数据库中所有的可用对象；后者可让你添加、更改和删除数据库中的某条记录。

其它语言

如果你的母语不是英语，而你不想用它来配置你的浏览器，你可以做一个快速更改来观察Django管理工具是否被翻译成你想要的语言。 仅需添加 `'django.middleware.locale.LocaleMiddleware'` 到 `MIDDLEWARE_CLASSES` 设置中，并确保它在'django.contrib.sessions.middleware.SessionMiddleware' 之后 。 （见上）

完成后，请刷新页面。 如果你设置的语言可用，一系列的链接文字将被显示成这种语言。这些文字包括页面顶端的Change password和Log out，页面中部的Groups和Users。 Django自带了多种语言的翻译。

关于Django更多的国际化特性，请参见第十九章。

点击Uers行中的Change链接，引导用户更改列表。

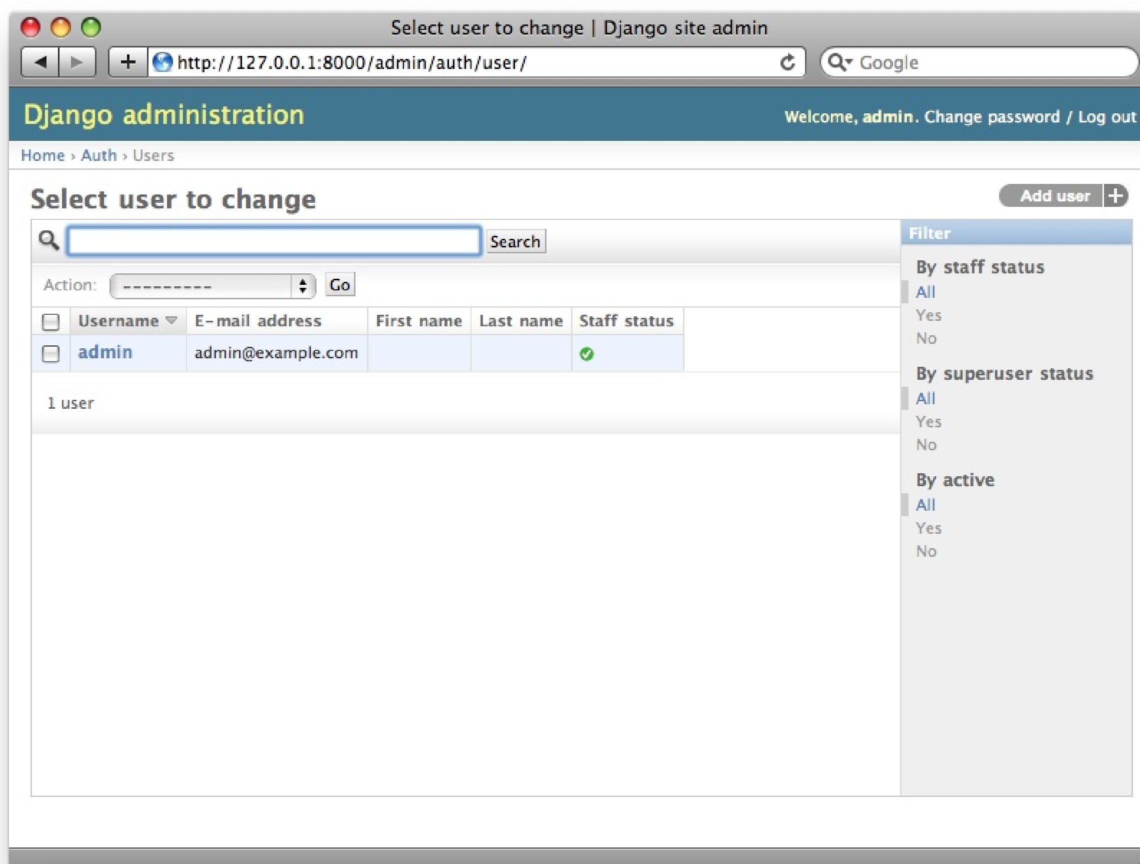


图 6-3. 典型的改变列表视图 （见上）

这个页面显示了数据库中所有的用户。你可以将它看作是一个漂亮的网页版查询： `SELECT * FROM auth_user;` 如果你一直跟着作练习，并且只添加了一个用户，你会在这个页面中看到一个用户。但是如果你添加了多个用户，你会发现页面中还有过滤器、排序和查询框。 过滤器在右边；排序功能可通过点击列头查看；查询框在页面顶部，它允许你通过用户名查询。

点击其中一个用户名，你会看见关于这个用户的编辑窗口。

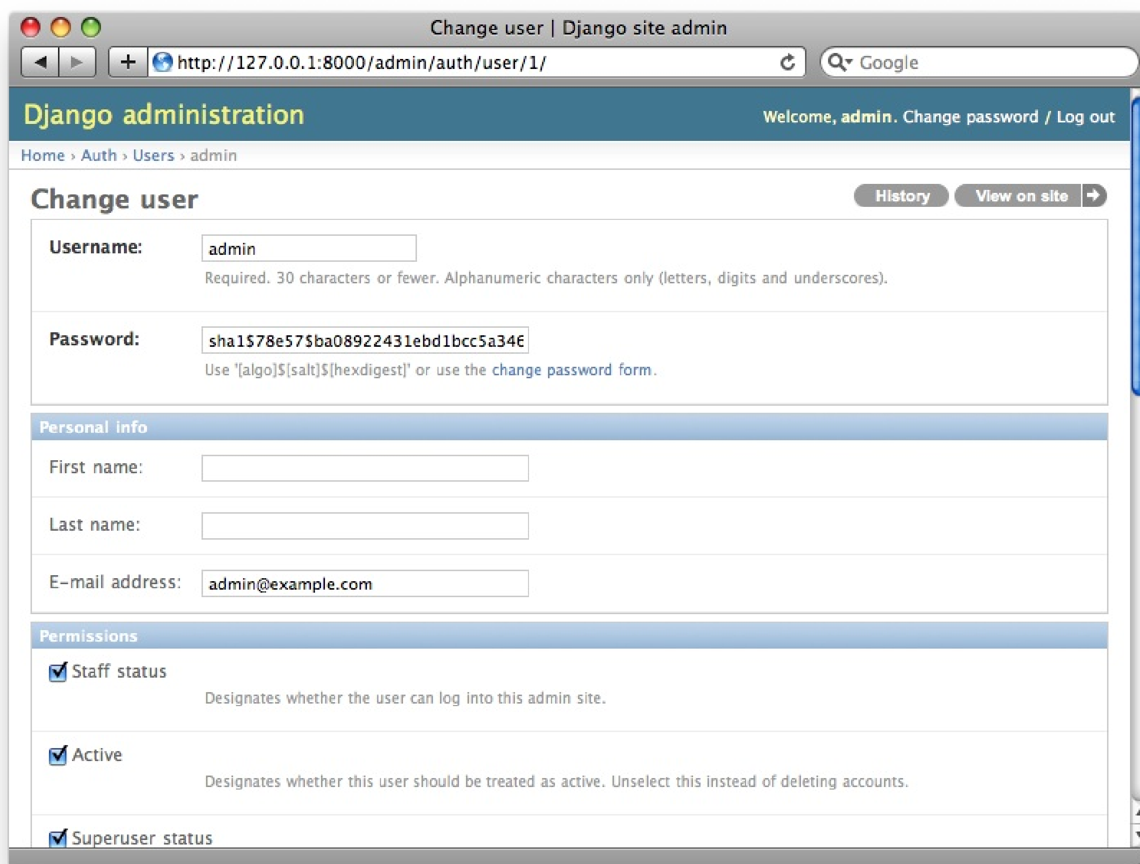


图 6-4. 典型的编辑表格 （见上）

这个页面允许你修改用户的属性，如姓名和权限。 （如果要更改用户密码，你必须点击密码字段下的change password form，而不是直接更改字段值中的哈西码。）另外需要注意的是，不同类型的字段会用不同的窗口控件显示。例如，日期/时间型用日历控件，布尔型用复选框，字符型用简单文本框显示。

你可以通过点击编辑页面下方的删除按钮来删除一条记录。 你会见到一个确认页面。有时候，它会显示有哪些关联的对象将会一并被删除。 （例如，如果你要删除一个出版社，它下面所有的图书也将被删除。）

你可以通过点击管理主页面中某个对象的Add来添加一条新记录。 一个空白记录的页面将被打开，等待你填充。

你还能看到管理界面也控制着你输入的有效性。 你可以试试不填必需的栏目或者在时间栏里填错误的时间，你会发现当你要保存时会出现错误信息，如图6-5所示。

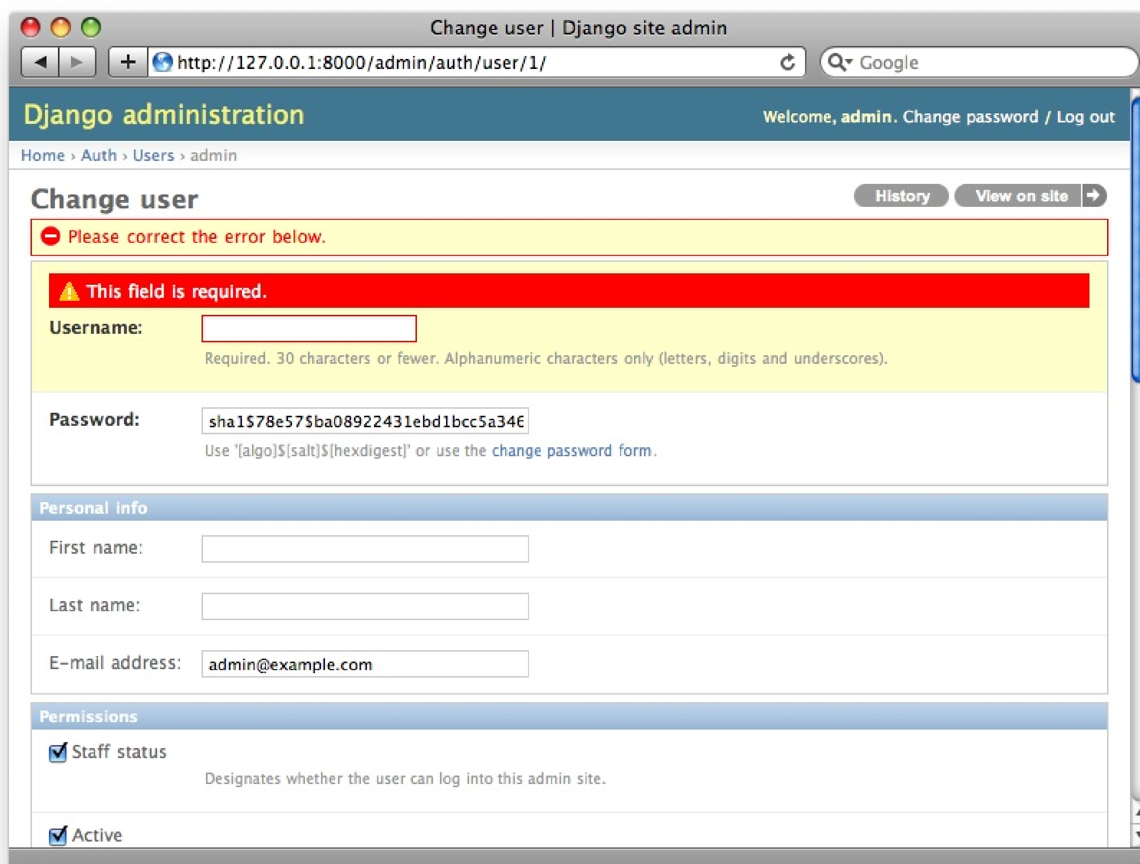


图6-5. 编辑表格显示错误信息 （见上）

当你编辑已有的对像时，你在窗口的右上角可以看到一个历史按钮。 通过管理界面做的每一个改变都留有记录，你可以按历史键来检查这个记录（见图6-6）。

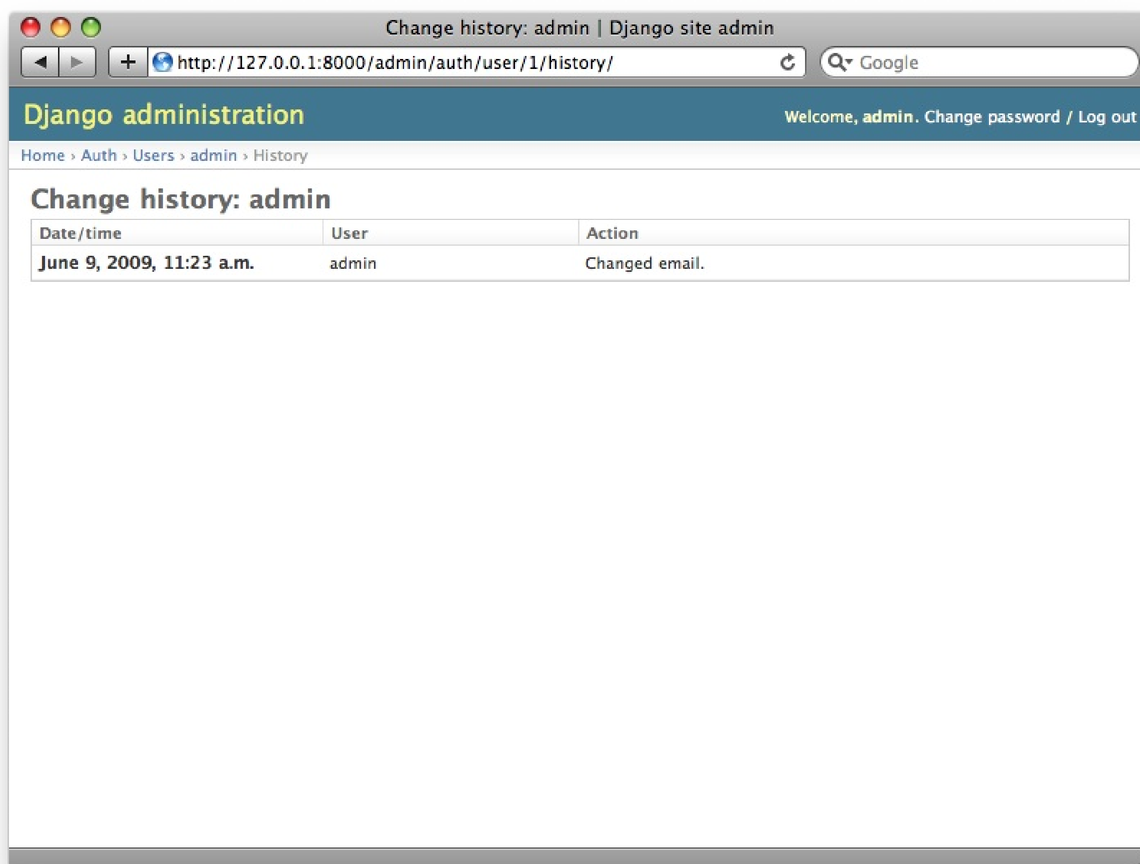


图6-6. Django 对像历史页面 （见上）

将你的Models加入到Admin管理中

有一个关键步骤我们还没做。 让我们将自己的模块加入管理工具中，这样我们就能够通过这个漂亮的界面添加、修改和删除数据库中的对象了。 我们将继续第五章中的 `book` 例子。在其中，我们定义了三个模块： `Publisher` 、 `Author` 和 `Book` 。

在 `books` 目录下(`mysite/books`)，创建一个文件： `admin.py` ，然后输入以下代码：

```
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

admin.site.register(Publisher)
admin.site.register(Author)
admin.site.register(Book)
```

这些代码通知管理工具为这些模块逐一提供界面。

完成后，打开页面 `http://127.0.0.1:8000/admin/` ，你会看到一个Books区域，其中包含Authors、Books和Publishers。 （你可能需要先停止，然后再启动服务(`runserver`)，才能使其生效。）

现在你拥有一个功能完整的管理界面来管理这三个模块了。 很简单吧！

花点时间添加和修改记录，以填充数据库。 如果你跟着第五章的例子一起创建Publisher对象的话（并且没有删除），你会在列表中看到那些记录。

这里需要提到的一个特性是，管理工具处理外键和多对多关系（这两种关系可以在 `Book` 模块中找到）的方法。 作为提醒，这里有个 `Book` 模块的例子：

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    def __unicode__(self):
        return self.title
```

在Add book页面中（ `http://127.0.0.1:8000/admin/books/book/add/` ）， `外键` publisher用一个选择框显示， `多对多` 字段author用一个多选框显示。 点击两个字段后面的绿色加号，可以让你添加相关的记录。 举个例子，如果你点击Publisher后面的加号，你将会得到一个弹出窗口来添加一个publisher。 当你在那个窗口中成功创建了一个publisher后，Add book表单会自动把它更新到字段上去 花巧.

Admin是如何工作的

在幕后，管理工具是如何工作的呢？ 其实很简单。

当服务启动时，Django从 `url.py` 引导URLconf，然后执行 `admin.autodiscover()` 语句。 这个函数遍历 `INSTALLED_APPS` 配置，并且寻找相关的 `admin.py` 文件。 如果在指定的app目录下找到 `admin.py` ，它就执行其中的代码。

在 `books` 应用程序目录下的 `admin.py` 文件中，每次调用 `admin.site.register()` 都将那个模块注册到管理工具中。 管理工具只为那些明确注册了的模块显示一个编辑/修改的界面。

应用程序 `django.contrib.auth` 包含自身的 `admin.py` ，所以Users和Groups能在管理工具中自动显示。 其它的 `django.contrib` 应用程序，如 `django.contrib.redirects` ，其它从网上下在的第三方Django应用程序一样，都会自行添加到管理工具。

综上所述，管理工具其实就是一个Django应用程序，包含自己的模块、模板、视图和URLpatterns。 你要像添加自己的视图一样，把它添加到URLconf里面。 你可以在Django基本代码中的 `django/contrib/admin` 目录下，检查它的模板、视图和URLpatterns，但你不要尝试直接修改其中的任何代码，因为里面有很多地方可以让你自定义管理工具的工作方式。 （如果你确实想浏览Django管理工具的代码，请谨记它在读取关于模块的元数据过程中做了些不简单的工作，因此最好花些时间阅读和理解那些代码。）

设置字段可选

在摆弄了一会之后，你或许会发现管理工具有个限制：编辑表单需要你填写每一个字段，然而在有些情况下，你想要某些字段是可选的。 举个例子，我们想要 `Author` 模块中的 `email` 字段成为可选，即允许不填。 在现实世界中，你可能没有为每个作者登记邮箱地址。

为了指定 `email` 字段为可选，你只要编辑 `Book` 模块（回想第五章，它在 `mysite/books/models.py` 文件里），在 `email` 字段上加上 `blank=True` 。代码如下：

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(**blank=True** )
```

这些代码告诉Django，作者的邮箱地址允许输入一个空值。 所有字段都默认 `blank=False` ，这使得它们不允许输入空值。

这里会发生一些有趣的事情。 直到现在，除了 `__unicode__()` 方法，我们的模块充当数据库中表定义的角色，即本质上是用Python的语法来写 `CREATE TABLE` 语句。 在添加 `blank=True` 过程中，我们已经开始在简单的定义数据表上扩展我们的模块了。 现在，我们的模块类开始成为一个富含 `Author` 对象属性和行为的集合了。 `email` 不但展现为一个数据库中的 `VARCHAR` 类型的字段，它还是页面中可选的字段，就像在管理工具中看到的那样。

当你添加 `blank=True` 以后，刷新页面Add author edit form (`http://127.0.0.1:8000/admin/books/author/add/`)，将会发现Email的标签不再是粗体了。 这意味它不是一个必填字段。 现在你可以添加一个作者而不必输入邮箱地址，即使你为这个字段提交了一个空值，也再不会得到那刺眼的红色信息“This field is required”。

设置日期型和数字型字段可选

虽然 `blank=True` 同样适用于日期型和数字型字段，但是这里需要详细讲解一些背景知识。

SQL有指定空值的独特方式，它把空值叫做NULL。NULL可以表示为未知的、非法的、或其它程序指定的含义。

在SQL中， `NULL` 的值不同于空字符串，就像Python中 `None` 不同于空字符串（ `""` ）一样。这意味着某个字符型字段（如 `VARCHAR` ）的值不可能同时包含 `NULL` 和空字符串。

这会引起不必要的歧义或疑惑。 为什么这条记录有个 `NULL` ，而那条记录却有个空字符串？它们之间有区别，还是数据输入不一致？ 还有： 我怎样才能得到全部拥有空值的记录，应该按 `NULL` 和空字符串查找么？还是仅按字符串查找？

为了消除歧义，Django生成 `CREATE TABLE` 语句自动为每个字段显式加上 `NOT NULL` 。 这里有个第五章中生成 `Author` 模块的例子：

```
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL
)
;
```

在大多数情况下，这种默认的行为对你的应用程序来说是最佳的，因为它可以使你不再因数据一致性而头痛。 而且它可以和Django的其它部分工作得很好。如在管理工具中，如果你留空一个字符型字段，它会为此插入一个空字符串（而 不是 `NULL` ）。

但是，其它数据类型有例外：日期型、时间型和数字型字段不接受空字符串。 如果你尝试将一个空字符串插入日期型或整数型字段，你可能会得到数据库返回的错误，这取决于那个数据库的类型。 （PostgreSQL比较严禁，会抛出一个异常；MySQL可能会也可能不会接受，这取决于你使用的版本和运气了。）在这种情况下， `NULL` 是唯一指定空值的方法。 在Django模块中，你可以通过添加 `null=True` 来指定一个字段允许为 `NULL` 。

因此，这说起来有点复杂： 如果你想允许一个日期型（ `DateField` 、 `TimeField` 、 `DateTimeField` ）或数字型（ `IntegerField` 、 `DecimalField` 、 `FloatField` ）字段为空，你需要使用 `null=True` 和 `blank=True` 。

为了举例说明，让我们把 `Book` 模块修改成允许 `publication_date` 为空。修改后的代码如下：

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField(**blank=True, null=True** )
```

添加 `null=True` 比添加 `blank=True` 复杂。因为 `null=True` 改变了数据的语义，即改变了 `CREATE TABLE` 语句，把 `publication_date` 字段上的 `NOT NULL` 删除了。 要完成这些改动，我们还需要更新数据库。

出于某种原因，Django不会尝试自动更新数据库结构。所以你必须执行 `ALTER TABLE` 语句将模块的改动更新至数据库。 像先前那样，你可以使用 `manage.py dbshell` 进入数据库服务环境。 以下是在这个特殊情况下如何删除 `NOT NULL` :

```
ALTER TABLE books_book ALTER COLUMN publication_date DROP NOT NULL;
```

（注意：以下SQL语法是PostgreSQL特有的。）

我们将在第十章详细讲述数据库结构更改。

现在让我们回到管理工具，添加book的编辑页面允许输入一个空的publication date。

自定义字段标签

在编辑页面中，每个字段的标签都是从模块的字段名称生成的。 规则很简单： 用空格替换下划线；首字母大写。例如： `Book` 模块中 `publication_date` 的标签是Publication date。

然而，字段名称并不总是贴切的。有些情况下，你可能想自定义一个标签。 你只需在模块中指定 `verbose_name` 。

举个例子，说明如何将 `Author.email` 的标签改为e-mail，中间有个横线。

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True, **verbose_name='e-mail'** )
```

修改后重启服务器，你会在author编辑页面中看到这个新标签。

请注意，你不必把 `verbose_name` 的首字母大写，除非是连续大写（如： `"USA state"` ）。 Django会自动适时将首字母大写，并且在其它不需要大写的地方使用 `verbose_name` 的精确值。

最后还需注意的是，为了使语法简洁，你可以把它当作固定位置的参数传递。 这个例子与上面那个的效果相同。

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(**'e-mail',**  blank=True)
```

但这不适用于 `ManyToManyField` 和 `ForeignKey` 字段，因为它们第一个参数必须是模块类。 那种情形，必须显式使用 `verbose_name` 这个参数名称。

自定义ModelAdmi类

迄今为止，我们做的 `blank=True` 、 `null=True` 和 `verbose_name` 修改其实是模块级别，而不是管理级别的。 也就是说，这些修改实质上是构成模块的一部分，并且正好被管理工具使用，而不是专门针对管理工具的。

除了这些，Django还提供了大量选项让你针对特别的模块自定义管理工具。 这些选项都在*ModelAdmin classes*里面，这些类包含了管理工具中针对特别模块的配置。

自定义列表

让我们更深一步：自定义 `Author` 模块的列表中的显示字段。 列表默认地显示查询结果中对象的 `__unicode__()` 。 在第五章中，我们定义 `Author` 对象的 `__unicode__()` 方法，用以同时显示作者的姓和名。

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField(blank=True, verbose_name='e-mail')

    **def __unicode__(self):**
        **return u'%s %s' % (self.first_name, self.last_name)**
```

结果正如图6-7所示，列表中显示的是每个作者的姓名。

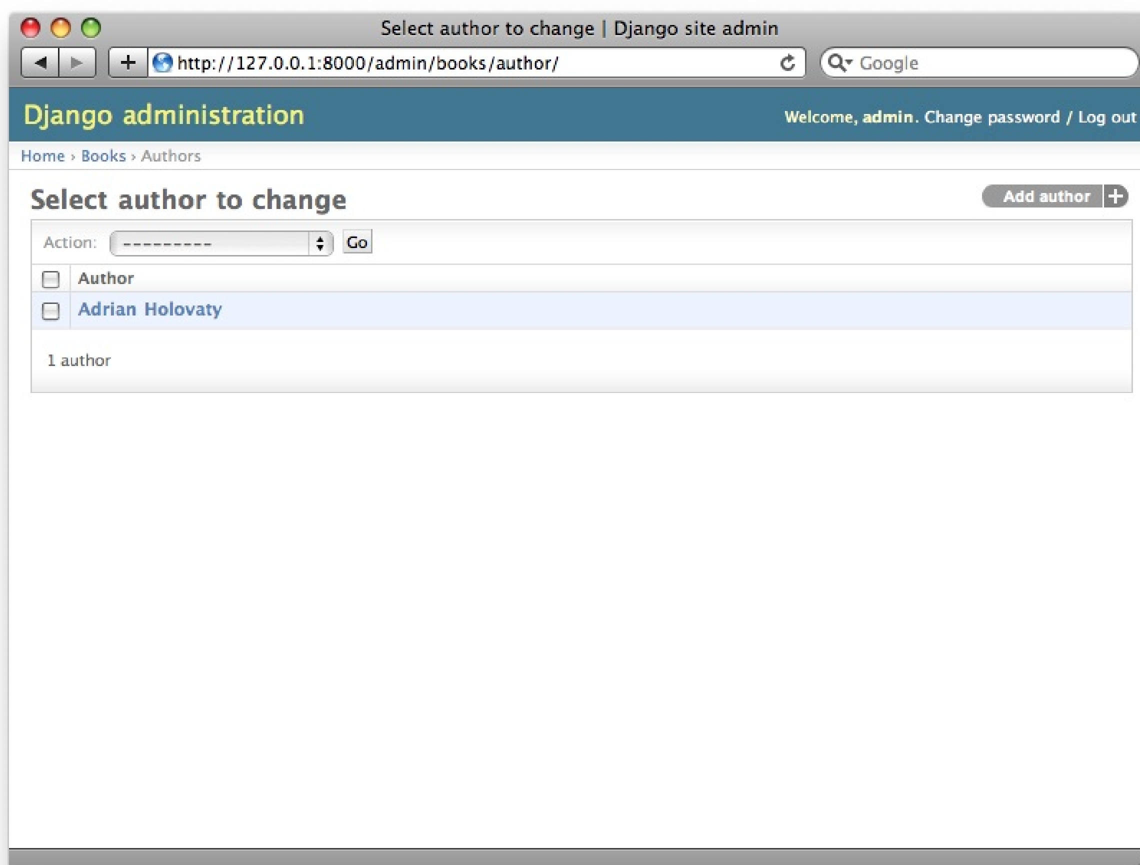


图 6-7. 作者列表

我们可以在这基础上改进，添加其它字段，从而改变列表的显示。 这个页面应该提供便利，比如说：在这个列表中可以看到作者的邮箱地址。如果能按照姓氏或名字来排序，那就更好了。

为了达到这个目的，我们将为 `Author` 模块定义一个 `ModelAdmin` 类。 这个类是自定义管理工具的关键，其中最基本的一件事情是允许你指定列表中的字段。 打开 `admin.py` 并修改：

```
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

**class AuthorAdmin(admin.ModelAdmin):**
    **list_display = ('first_name', 'last_name', 'email')**

admin.site.register(Publisher)
**admin.site.register(Author, AuthorAdmin)**
admin.site.register(Book)
```

解释一下代码：

> 我们新建了一个类 `AuthorAdmin`， 它是从 `django.contrib.admin.ModelAdmin` 派生出来的子类，保存着一个类的自定义配置，以供管理工具使用。 我们只自定义了一项： `list_display`， 它是一个字段名称的元组，用于列表显示。 当然，这些字段名称必须是模块中有的。
>
> 我们修改了 `admin.site.register()` 调用，在 `Author` 后面添加了 `AuthorAdmin` 。你可以这样理解： 用 `AuthorAdmin` 选项注册 `Author` 模块。
>
> `admin.site.register()` 函数接受一个 `ModelAdmin` 子类作为第二个参数。 如果你忽略第二个参数，Django将使用默认的选项。 `Publisher` 和 `Book` 的注册就属于这种情况。

弄好了这个东东，再刷新author列表页面，你会看到列表中有三列：姓氏、名字和邮箱地址。另外，点击每个列的列头可以对那列进行排序。 （参见图 6-8）

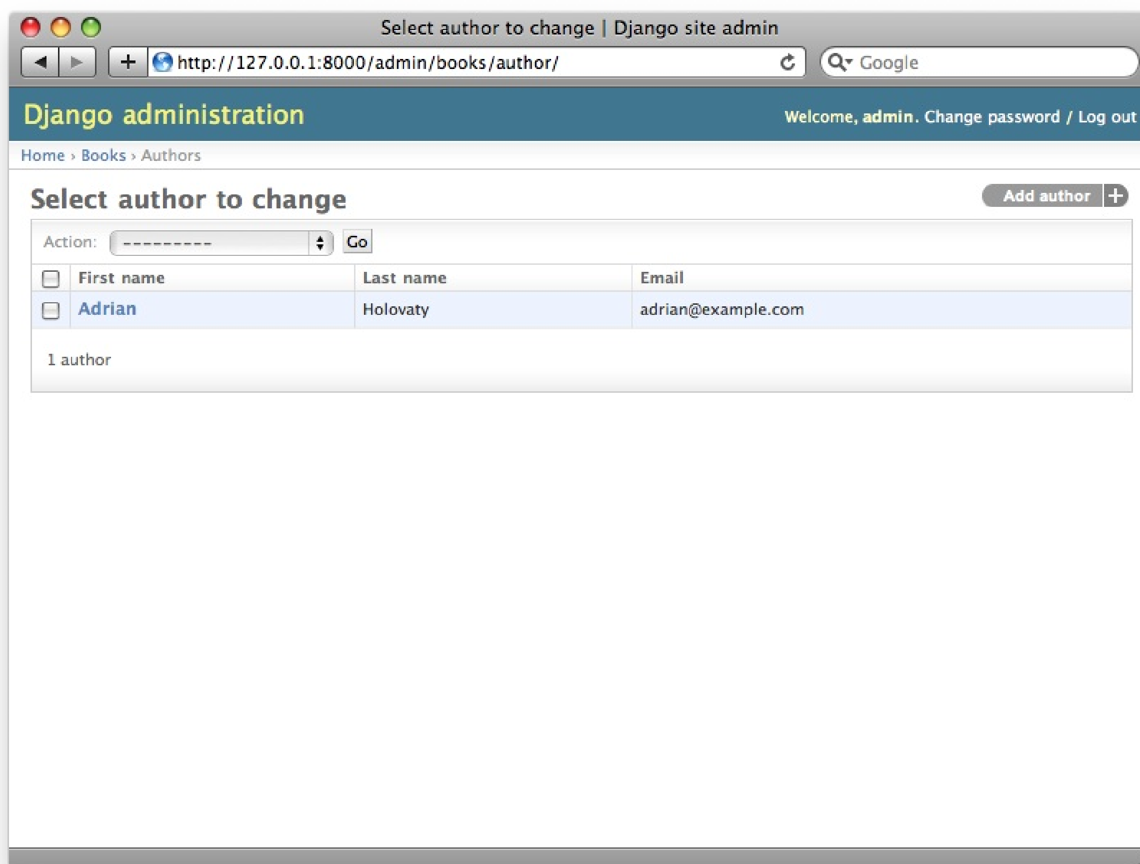


图 6-8. 修改后的author列表页面

接下来，让我们添加一个快速查询栏。 向 `AuthorAdmin` 追加 `search_fields` ， 如：

```
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    **search_fields = ('first_name', 'last_name')**
```

刷新浏览器，你会在页面顶端看到一个查询栏。 （见图6-9.）我们刚才所作的修改列表页面，添加了一个根据姓名查询的查询框。 正如用户所希望的那样，它是大小写敏感，并且对两个字段检索的查询框。如果查询 `"bar"` ，那么名字中含有Barney和姓氏中含有Hobarson的作者记录将被检索出来。

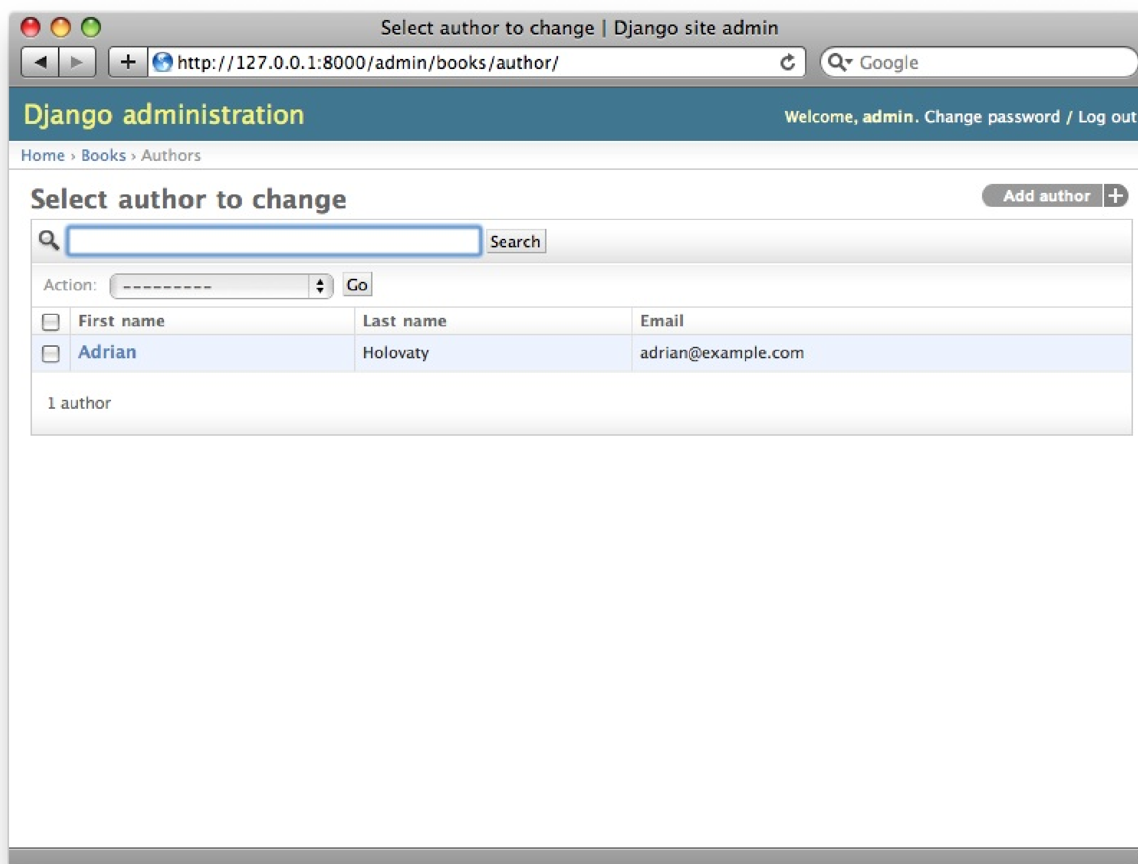


图 6-9. 含search_fields的author列表页面

接下来，让我们为 `Book` 列表页添加一些过滤器。

```
from django.contrib import admin
from mysite.books.models import Publisher, Author, Book

class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    search_fields = ('first_name', 'last_name')

**class BookAdmin(admin.ModelAdmin):**
    **list_display = ('title', 'publisher', 'publication_date')**
    **list_filter = ('publication_date',)**

admin.site.register(Publisher)
admin.site.register(Author, AuthorAdmin)
**admin.site.register(Book, BookAdmin)**
```

由于我们要处理一系列选项，因此我们创建了一个单独的 `ModelAdmin` 类： `BookAdmin` 。首先，我们定义一个 `list_display` ，以使得页面好看些。 然后，我们用 `list_filter` 这个字段元组创建过滤器，它位于列表页面的右边。 Django为日期型字段提供了快捷过滤方式，它包含：今天、过往七天、当月和今年。这些是开发人员经常用到的。 图 6-10显示了修改后的页面。

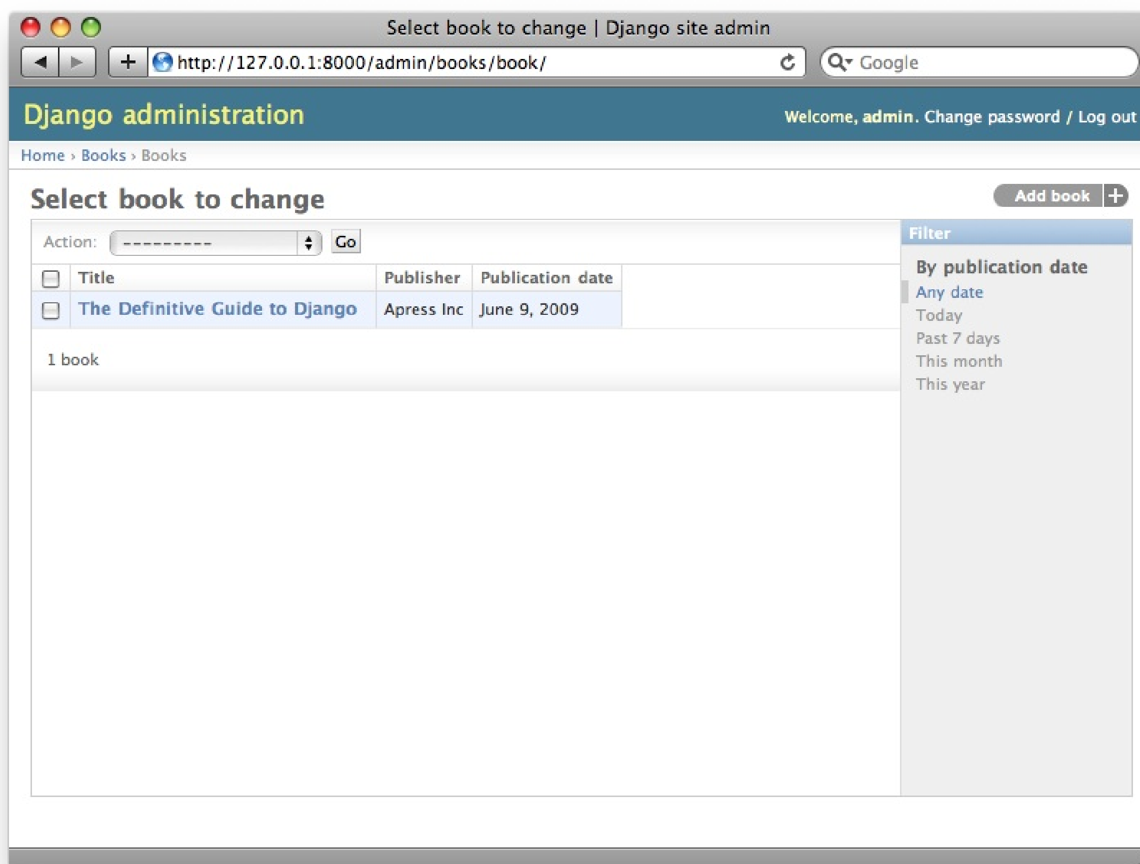


图 6-10. 含过滤器的book列表页面

`过滤器` 同样适用于其它类型的字段，而不单是 `日期型` （请在 `布尔型` 和 `外键` 字段上试试）。当有两个以上值时，过滤器就会显示。

另外一种过滤日期的方式是使用 `date_hierarchy` 选项，如：

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    **date_hierarchy = 'publication_date'**
```

修改好后，页面中的列表顶端会有一个逐层深入的导航条，效果如图 6-11. 它从可用的年份开始，然后逐层细分到月乃至日。

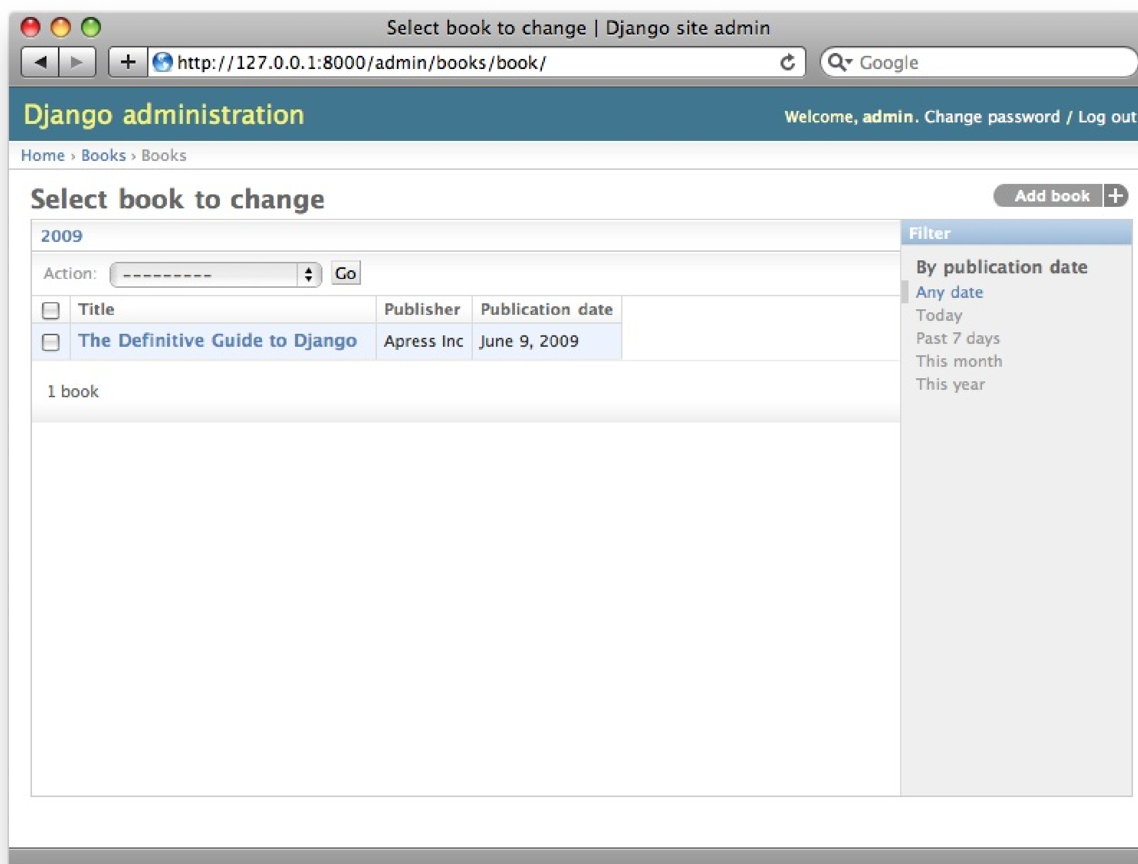


图 6-11. 含date_hierarchy的book列表页面

请注意， `date_hierarchy` 接受的是 字符串 ，而不是元组。因为只能对一个日期型字段进行层次划分。

最后，让我们改变默认的排序方式，按publication date降序排列。 列表页面默认按照模块 `class Meta` （详见第五章）中的 `ordering` 所指的列排序。但目前没有指定 `ordering` 值，所以当前排序是没有定义的。

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    **ordering = ('-publication_date',)**
```

这个 `ordering` 选项基本像模块中 `class Meta` 的 `ordering` 那样工作，除了它只用列表中的第一个字段名。 如果要实现降序，仅需在传入的列表或元组的字段前加上一个减号(-)。

刷新book列表页面观看实际效果。 注意Publication date列头现在有一个小箭头显示排序。（见图 6-12.）

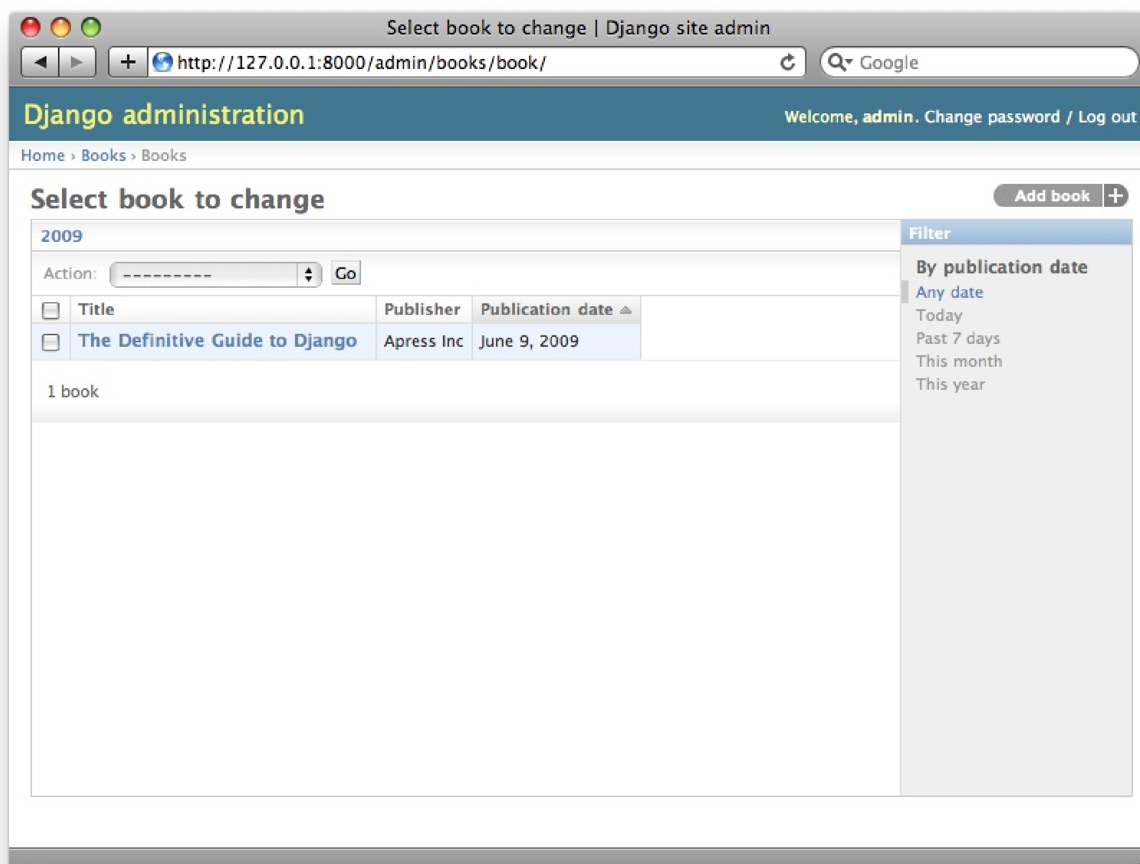


图 6-12 含排序的book列表页面

我们已经学习了主要的选项。 通过使用它们，你可以仅需几行代码就能创建一个功能强大、随时上线的数据编辑界面。

自定义编辑表单

正如自定义列表那样，编辑表单多方面也能自定义。

首先，我们先自定义字段顺序。 默认地，表单中的字段顺序是与模块中定义是一致的。 我们可以通过使用 `ModelAdmin` 子类中的 `fields` 选项来改变它：

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    **fields = ('title', 'authors', 'publisher', 'publication_date')**
```

完成之后，编辑表单将按照指定的顺序显示各字段。 它看起来自然多了——作者排在书名之后。 字段顺序当然是与数据条目录入顺序有关， 每个表单都不一样。

通过fields这个选项，你可以排除一些不想被其他人编辑的fields 只要不选上不想被编辑的field(s)即可。 当你的admi用户只是被信任可以更改你的某一部分数据时，或者，你的数据被一些外部的程序自动处理而改变了了，你就可以用这个功能。 例如，在book数据库中，我们可以隐藏`publication_date`，以防止它被编辑。

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    **fields = ('title', 'authors', 'publisher')**
```

这样，在编辑页面就无法对publication date进行改动。 如果你是一个编辑，不希望作者推迟出版日期的话，这个功能就很有用。 （当然，这纯粹是一个假设的例子。）

当一个用户用这个不包含完整信息的表单添加一本新书时，Django会简单地将`publication_date`设置为`None`，以确保这个字段满足`null=True`的条件。

另一个常用的编辑页面自定义是针对多对多字段的。 真如我们在book编辑页面看到的那样，`多对多字段`被展现成多选框。虽然多选框在逻辑上是最适合的HTML控件，但它却不那么好用。 如果你想选择多项，你必须还要按下Ctrl键（苹果机是command键）。 虽然管理工具因此添加了注释（help_text），但是当它有几百个选项时，它依然显得笨拙。

更好的办法是使用`filter_horizontal`。让我们把它添加到`BookAdmin`中，然后看看它的效果。

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    **filter_horizontal = ('authors',)**
```

（如果你一着跟着做练习，请注意移除`fields`选项，以使得编辑页面包含所有字段。）

刷新book编辑页面，你会看到Author区中有一个精巧的JavaScript过滤器，它允许你检索选项，然后将选中的authors从Available框移到Chosen框，还可以移回来。

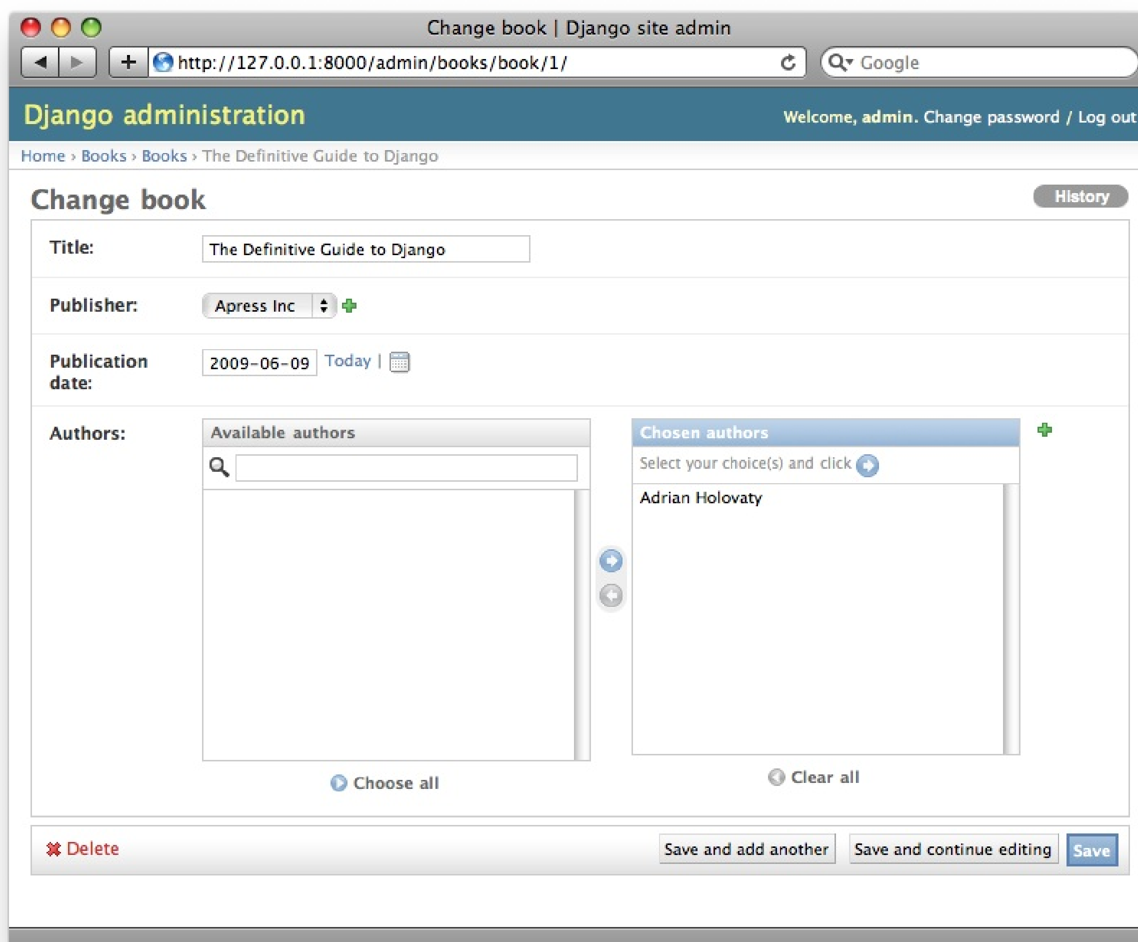


图 6-13. 含filter_horizontal的book编辑页面

我们强烈建议针对那些拥有十个以上选项的 `多对多字段` 使用 `filter_horizontal` 。 这比多选框好用多了。 你可以在多个字段上使用 `filter_horizontal` ，只需在这个元组中指定每个字段的名字。

`ModelAdmin` 类还支持 `filter_vertical` 选项。 它像 `filter_horizontal` 那样工作，除了控件都是垂直排列，而不是水平排列的。 至于使用哪个，只是个人喜好问题。

`filter_horizontal` 和 `filter_vertical` 选项只能用在 `多对多字段` 上, 而不能用于 `ForeignKey` 字段。 默认地，管理工具使用 `下拉框` 来展现 `外键` 字段。但是，正如 `多对多字段` 那样，有时候你不想忍受因装载并显示这些选项而产生的大量开销。 例如，我们的book数据库膨胀到拥有数千条publishers的记录，以致于book的添加页面装载时间较久，因为它必须把每一个publishe都装载并显示在 `下拉框` 中。

解决这个问题的办法是使用 `raw_id_fields` 选项。它是一个包含外键字段名称的元组，它包含的字段将被展现成 `文本框` ，而不再是 `下拉框` 。 见图 6-14。

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
    list_filter = ('publication_date',)
    date_hierarchy = 'publication_date'
    ordering = ('-publication_date',)
    filter_horizontal = ('authors',)
    **raw_id_fields = ('publisher',)**
```

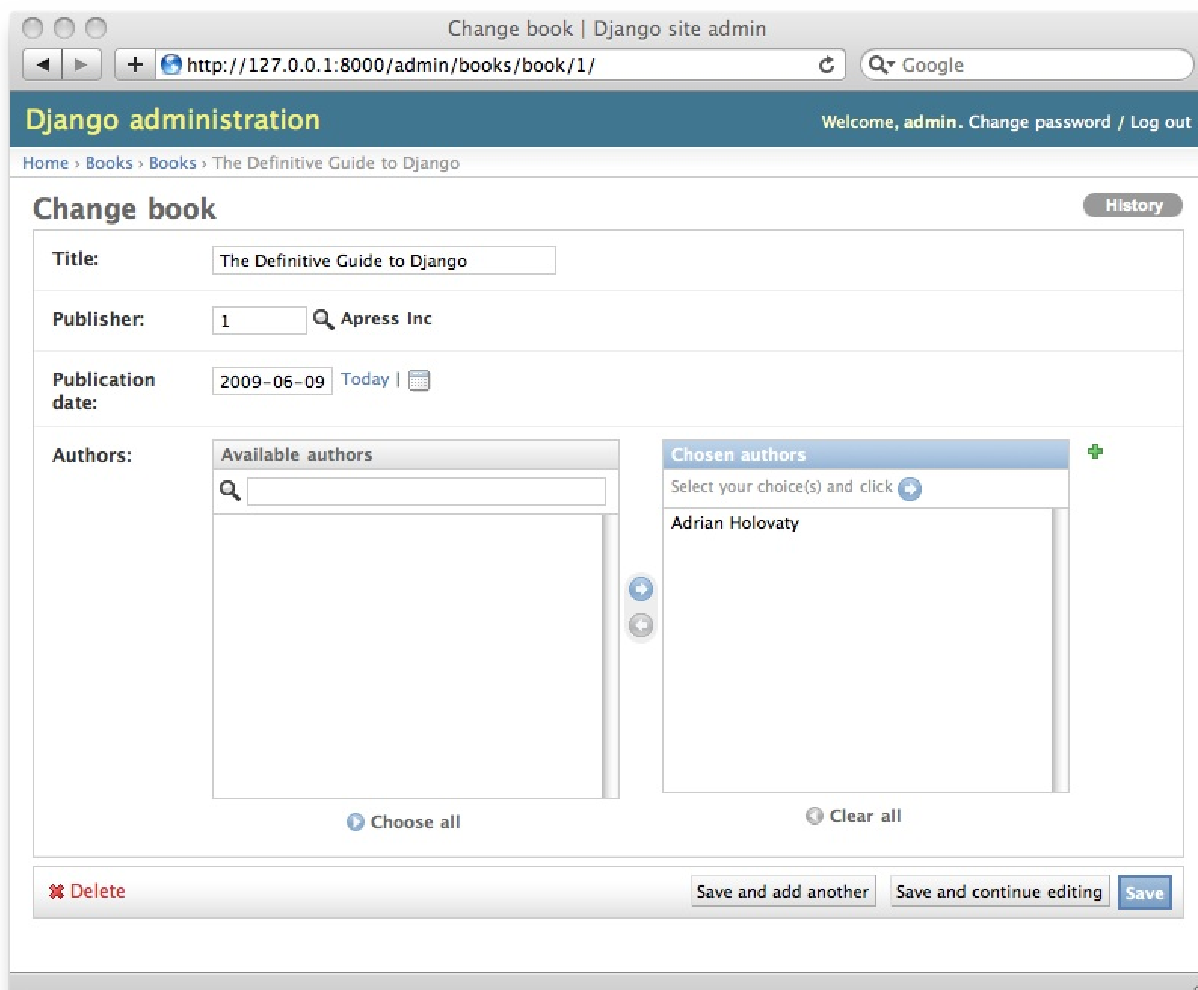


图 6-14. 含raw_id_fields的book编辑页面

在这个输入框中，你输入什么呢？ publisher的数据库ID号。 考虑到人们通常不会记住这些数据库ID，管理工具提供了一个放大镜图标方便你输入。点击那个图标将会弹出一个窗口，在那里你可以选择想要添加的publishe。

用户、用户组和权限

因为你是用超级用户登录的，你可以创建，编辑和删除任何对像。 然而，不同的环境要求有不同的权限，系统不允许所有人都是超级用户。 管理工具有一个用户权限系统，通过它你可以根据用户的需要来指定他们的权限，从而达到部分访问系统的目的。

用户帐号应该是通用的、独立于管理界面以外仍可以使用。但我们现在把它看作是管理界面的一部分。 在第十四章，我们将讲述如何把用户帐号与你的网站（不仅仅是管理工具）集成在一起。

你通过管理界面编辑用户及其许可就像你编辑别的对象一样。 我们在本章的前面，浏览用户和用户组区域的时候已经见过这些了。 如你所想，用户对象有标准的用户名、密码、邮箱地址和真实姓名，同时它还有关于使用管理界面的权限定义。 首先，这有一组三个布尔型标记：

- 活动标志，它用来控制用户是否已经激活。 如果一个用户帐号的这个标记是关闭状态，而用户又尝试用它登录时，即使密码正确，他也无法登录系统。
- 成员标志，它用来控制这个用户是否可以登录管理界面（即：这个用户是不是你们组织里的成员） 由于用户系统可以被用于控制公众页面（即：非管理页面）的访问权限（详见第十四章），这个标志可用来区分公众用户和管理用户。
- 超级用户标志，它赋予用户在管理界面中添加、修改和删除任何项目的权限。 如果一个用户帐号有这个标志，那么所有权限设置（即使没有）都会被忽略。

普通的活跃，非超级用户的管理用户可以根据一套设定好的许可进入。 管理界面中每种可编辑的对象（如：books、authors、publishers）都有三种权限： 创建 许可， 编辑 许可和 删除 许可。 给一个用户授权许可也就表明该用户可以进行许可描述的操作。

当你创建一个用户时，它没有任何权限，该有什么权限是由你决定的。 例如，你可以给一个用户添加和修改publishers的权限，而不给他删除的权限。 请注意，这些权限是定义在模块级别上，而不是对象级别上的。据个例子，你可以让小强修改任何图书，但是不能让他仅修改由机械工业出版社出版的图书。 后面这种基于对象级别的权限设置比较复杂，并且超出了本书的覆盖范围，但你可以在Django documentation中寻找答案。

注释

权限管理系统也控制编辑用户和权限。 如果你给某人编辑用户的权限，他可以编辑自己的权限，这种能力可能不是你希望的。 赋予一个用户修改用户的权限，本质上说就是把他变成一个超级用户。

你也可以给组中分配用户。 一个 组 简化了给组中所有成员应用一套许可的动作。 组在给大量用户特定权限的时候很有用。

何时、为什么使用管理界面？何时又不使用呢？

经过这一章的学习，你应该对Django管理工具有所认识。 但是我们需要表明一个观点： 什么时候 、 为什么 用，以及什么时候又 不 用。

Django的管理界面对非技术用户要输入他们的数据时特别有用；事实上这个特性就是专门为这个 实现的。 在Django最开始开发的新闻报道的行业应用中，有一个典型的在线自来水的水质专题报道 应用，它的实现流程是这样的：

- 负责这个报道的记者和要处理数据的开发者碰头，提供一些数据给开发者。
- 开发者围绕这些数据设计模型然后配置一个管理界面给记者。
- 记者检查管理界面，尽早指出缺少或多余的字段。 开发者来回地修改模块。
- 当模块认可后，记者就开始用管理界面输入数据。 同时，程序员可以专注于开发公众访问视图和模板（有趣的部分）。

换句话说，Django的管理界面为内容输入人员和编程人员都提供了便利的工具。

当然，除了数据输入方面，我们发现管理界面在下面这些情景中也是很有用的：

-
    - 检查模块* ：当你定义好了若干个模块，在管理页面中把他们调出来然后输入一些虚假的数据，这是相当有用的。 有时候，它能显示数据建模的错误或者模块中其它问题。
-
    - 管理既得数据* ：如果你的应用程序依赖外部数据（来自用户输入或网络爬虫），管理界面提供了一个便捷的途径，让你检查和编辑那些数据。 你可以把它看作是一个功能不那么强大，但是很方便的数据库命令行工具。
-
    - 临时的数据管理程序* ：你可以用管理工具建立自己的轻量级数据管理程序，比如说开销记录。 如果你正在根据自己的，而不是公众的需要开发些什么，那么管理界面可以带给你很大的帮助。 从这个意义上讲，你可以把它看作是一个增强的关系型电子表格。

最后一点要澄清的是： 管理界面不是终结者。 过往许多年间，我们看到它被拆分、修改成若干个功能模块，而这些功能不是它所支持的。 它不应成为一个 公众 数据访问接口，也不应允许对你的数据进行复杂的排序和查询。 正如本章开头所说，它仅提供给可信任的管理员。 请记住这一点，它是有效使用管理界面的钥匙。

下一章

到现在，我们已经创建了一些模块，并且为编辑数据配置了一个优秀的界面。下一章，我们将转入到网站开发中最重要的部分： 表单的创建和处理。

第七章：表单

从Google的简朴的单个搜索框，到常见的Blog评论提交表单，再到复杂的自定义数据输入接口，HTML表单一直是交互性网站的支柱。 本章介绍如何用Django对用户通过表单提交的数据进行访问、有效性检查以及其它处理。 与此同时，我们将介绍HttpRequest对象和Form对象。

从Request对象中获取数据

我们在第三章讲述View的函数时已经介绍过HttpRequest对象了，但当时并没有讲太多。 让我们回忆下：每个view函数的第一个参数是一个HttpRequest对象，就像下面这个hello()函数:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

HttpRequest对象，比如上面代码里的request变量，会有一些有趣的、你必须让自己熟悉的属性和方法，以便知道能拿它们来做些什么。 在view函数的执行过程中，你可以用这些属性来获取当前request的一些信息（比如，你正在加载这个页面的用户是谁，或者用的是什么浏览器）。

URL相关信息

HttpRequest对象包含当前请求URL的一些信息：

属性/方法	说明	举例
`request.path`	除域名以外的请求路径，以正斜杠开头	`"/hello/"`
`request.get_host()`	主机名（比如，通常所说的域名）	`"127.0.0.1:8000"` or `"www.example.com"`
`request.get_full_path()`	请求路径，可能包含查询字符串	`"/hello/?print=true"`
`request.is_secure()`	如果通过HTTPS访问，则此方法返回True， 否则返回False	`True` 或者 `False`

在view函数里，要始终用这个属性或方法来得到URL，而不要手动输入。 这会使得代码更加灵活，以便在其它地方重用。 下面是一个简单的例子：

```
# BAD!
def current_url_view_bad(request):
    return HttpResponse("Welcome to the page at /current/")

# GOOD
def current_url_view_good(request):
    return HttpResponse("Welcome to the page at %s" % request.path)
```

有关request的其它信息

request.META 是一个Python字典，包含了所有本次HTTP请求的Header信息，比如用户IP地址和用户Agent（通常是浏览器的名称和版本号）。 注意，Header信息的完整列表取决于用户所发送的Header信息和服务器端设置的Header信息。 这个字典中几个常见的键值有：

- `HTTP_REFERER`，进站前链接网页，如果有的话。 （请注意，它是 `REFERRER` 的笔误。）
- `HTTP_USER_AGENT`，用户浏览器的user-agent字符串，如果有的话。 例如： `"Mozilla/5.0 (X11; U; Linux i686; fr-FR; rv:1.8.1.17) Gecko/20080829 Firefox/2.0.0.17"` .
- `REMOTE_ADDR` 客户端IP，如：`"12.345.67.89"` 。(如果申请是经过代理服务器的话，那么它可能是以逗号分割的多个IP地址，如：`"12.345.67.89,23.456.78.90"` 。)

注意，因为 request.META 是一个普通的Python字典，因此当你试图访问一个不存在的键时，会触发一个KeyError异常。 （HTTP header信息是由用户的浏览器所提交的、不应该给予信任的"额外"数据，因此你总是应该好好设计你的应用以便当一个特定的Header数据不存在时，给出一个优雅的回应。）你应该用 try/except 语句，或者用Python字典的 get() 方法来处理这些"可能不存在的键"：

```
# BAD!
def ua_display_bad(request):
    ua = request.META['HTTP_USER_AGENT']  # Might raise KeyError!
    return HttpResponse("Your browser is %s" % ua)

# GOOD (VERSION 1)
def ua_display_good1(request):
    try:
        ua = request.META['HTTP_USER_AGENT']
    except KeyError:
        ua = 'unknown'
    return HttpResponse("Your browser is %s" % ua)

# GOOD (VERSION 2)
def ua_display_good2(request):
    ua = request.META.get('HTTP_USER_AGENT', 'unknown')
    return HttpResponse("Your browser is %s" % ua)
```

我们鼓励你动手写一个简单的view函数来显示 request.META 的所有数据，这样你就知道里面有什么了。 这个view函数可能是这样的：

```
def display_meta(request):
    values = request.META.items()
    values.sort()
    html = []
    for k, v in values:
        html.append('<tr><td>%s</td><td>%s</td></tr>' % (k, v))
    return HttpResponse('<table>%s</table>' % '\n'.join(html))
```

做为一个练习，看你自己能不能把上面这个view函数改用Django模板系统来实现，而不是上面这样来手动输入HTML代码。 也可以试着把前面提到的 request.path 方法或 HttpRequest 对象的其它方法加进去。

提交的数据信息

除了基本的元数据，HttpRequest对象还有两个属性包含了用户所提交的信息： request.GET 和 request.POST。二者都是类字典对象，你可以通过它们来访问GET和POST数据。

类字典对象

我们说"request.GET和request.POST是类字典对象"，意思是他们的行为像Python里标准的字典对象，但在技术底层上他们不是标准字典对象。 比如说，request.GET和request.POST都有get()、keys()和values()方法，你可以用用 for key in request.GET 获取所有的键。

那到底有什么区别呢？ 因为request.GET和request.POST拥有一些普通的字典对象所没有的方法。 我们会稍后讲到。

你可能以前遇到过相似的名字：类文件对象，这些Python对象有一些基本的方法，如read()，用来做真正的Python文件对象的代用品。

POST数据是来自HTML中的〈form〉标签提交的，而GET数据可能来自〈form〉提交也可能是URL中的查询字符串(the query string)。

一个简单的表单处理示例

继续本书一直进行的关于书籍、作者、出版社的例子，我们现在来创建一个简单的view函数以便让用户可以通过书名从数据库中查找书籍。

通常，表单开发分为两个部分： 前端HTML页面用户接口和后台view函数对所提交数据的处理过程。 第一部分很简单；现在我们来建立个view来显示一个搜索表单：

```
from django.shortcuts import render_to_response

def search_form(request):
    return render_to_response('search_form.html')
```

在第三章已经学过，这个view函数可以放到Python的搜索路径的任何位置。 为了便于讨论，咱们将它放在 books/views.py 里。

这个 search_form.html 模板，可能看起来是这样的：

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

而 urls.py 中的 URLpattern 可能是这样的：

```
from mysite.books import views

urlpatterns = patterns('',
    # ...
    (r'^search-form/$', views.search_form),
    # ...
)
```

（注意，我们直接将views模块import进来了，而不是用类似 from mysite.views import search_form 这样的语句，因为前者看起来更简洁。 我们将在第 8 章讲述更多的关于import的用法。）

现在，如果你运行 `runserver` 命令，然后访问 `http://127.0.0.1:8000/search-form/` ，你会看到搜索界面。 非常简单。

不过，当你通过这个form提交数据时，你会得到一个Django 404错误。 这个Form指向的URL `/search/` 还没有被实现。 让我们添加第二个视图函数并设置URL：

```
# urls.py

urlpatterns = patterns('',
    # ...
    (r'^search-form/$', views.search_form),
    (r'^search/$', views.search),
    # ...
)

# views.py

def search(request):
    if 'q' in request.GET:
        message = 'You searched for: %r' % request.GET['q']
    else:
        message = 'You submitted an empty form.'
    return HttpResponse(message)
```

暂时先只显示用户搜索的字词，以确定搜索数据被正确地提交给了Django，这样你就会知道搜索数据是如何在这个系统中传递的。 简而言之：

1. 在HTML里我们定义了一个变量q。当提交表单时，变量q的值通过GET(method=”get”)附加在URL /search/上。

2. 处理/search/（search()）的视图通过request.GET来获取q的值。

需要注意的是在这里明确地判断q是否包含在request.GET中。就像上面request.META小节里面提到，对于用户提交过来的数据，甚至是正确的数据，都需要进行过滤。 在这里若没有进行检测，那么用户提交一个空的表单将引发KeyError异常：

```
# BAD!
def bad_search(request):
    # The following line will raise KeyError if 'q' hasn't
    # been submitted!
    message = 'You searched for: %r' % request.GET['q']
    return HttpResponse(message)
```

查询字符串参数

因为使用GET方法的数据是通过查询字符串的方式传递的（例如/search/?q=django），所以我们可以使用requet.GET来获取这些数据。 第三章介绍Django的URLconf系统时我们比较了Django的简洁的URL与PHP/Java传统的URL，我们提到将在第七章讲述如何使用传统的URL。通过刚才的介绍，我们知道在视图里可以使用request.GET来获取传统URL里的查询字符串（例如hours=3）。

获取使用POST方法的数据与GET的相似，只是使用request.POST代替了request.GET。那么，POST与GET之间有什么不同？当我们提交表单仅仅需要获取数据时就可以用GET； 而当我们提交表单时需要更改服务器数据的状态，或者说发送e-mail，或者其他不仅仅是获取并显示数据的时候就使用POST。 在这个搜索书籍的例子里，我们使用GET，因为这个查询不会更改服务器数据的状态。 (如果你有兴趣了解更多关于 `GET` 和 `POST` 的知识，可以参见 http://www.w3.org/2001/tag/doc/whenToUseGet.html。)

既然已经确认用户所提交的数据是有效的，那么接下来就可以从数据库中查询这个有效的数据（同样，在views.py里操作）：

```
from django.http import HttpResponse
from django.shortcuts import render_to_response
from mysite.books.models import Book

def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render_to_response('search_results.html',
            {'books': books, 'query': q})
    else:
        return HttpResponse('Please submit a search term.')
```

让我们来分析一下上面的代码：

> 除了检查q是否存在于request.GET之外，我们还检查来reuqest.GET[‘q’]的值是否为空。
>
> 我们使用Book.objects.filter(title__icontains=q)获取数据库中标题包含q的书籍。icontains是一个查询关键字（参看第五章和附录B）。这个语句可以理解为获取标题里包含q的书籍，不区分大小写。
>
> 这是实现书籍查询的一个很简单的方法。 我们不推荐在一个包含大量产品的数据库中使用icontains查询，因为那会很慢。 （在真实的案例中，我们可以使用以某种分类的自定义查询系统。 在网上搜索“开源 全文搜索”看看是否有好的方法）
>
> 最后，我们给模板传递来books，一个包含Book对象的列表。 查询结果的显示模板search_results.html如下所示：

```
<p>You searched for: <strong>{{ query }}</strong></p>

{% if books %}
    <p>Found {{ books|length }} book{{ books|pluralize }}.</p>
    <ul>
        {% for book in books %}
        <li>{{ book.title }}</li>
        {% endfor %}
    </ul>
{% else %}
    <p>No books matched your search criteria.</p>
{% endif %}
```

> 注意这里pluralize的使用，这个过滤器在适当的时候会输出s（例如找到多本书籍）。

改进表单

同上一章一样，我们先从最为简单、有效的例子开始。 现在我们再来找出这个简单的例子中的不足，然后改进他们。

首先，search()视图对于空字符串的处理相当薄弱——仅显示一条”Please submit a search term.”的提示信息。 若用户要重新填写表单必须自行点击“后退”按钮， 这种做法既糟糕又不专业。如果在现实的案例中，我们这样子编写，那么Django的优势将荡然无存。

在检测到空字符串时更好的解决方法是重新显示表单，并在表单上面给出错误提示以便用户立刻重新填写。 最简单的实现方法既是添加else分句重新显示表单，代码如下：

```
from django.http import HttpResponse
from django.shortcuts import render_to_response
from mysite.books.models import Book

def search_form(request):
    return render_to_response('search_form.html')

def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render_to_response('search_results.html',
            {'books': books, 'query': q})
    else:
        **return render_to_response('search_form.html', {'error': True})**
```

（注意，将search_form()视图也包含进来以便查看）

这段代码里，我们改进来search()视图：在字符串为空时重新显示search_form.html。 并且给这个模板传递了一个变量error，记录着错误提示信息。 现在我们编辑一下search_form.html，检测变量error：

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    **{% if error %}**
        **<p style="color: red;">Please submit a search term.</p>**
    **{% endif %}**
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

我们修改了search_form()视图所使用的模板，因为search_form()视图没有传递error变量，所以在条用search_form视图时不会显示错误信息。

通过上面的一些修改，现在程序变的好多了，但是现在出现一个问题： 是否有必要专门编写search_form()来显示表单？ 按实际情况来说，当一个请求发送至/search/（未包含GET的数据）后将会显示一个空的表单（带有错误信息）。 所以，只要我们改变search()视图：当用户访问/search/并未提交任何数据时就隐藏错误信息，这样就移去search_form()视图以及对应的URLpattern。

```
def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        else:
            books = Book.objects.filter(title__icontains=q)
            return render_to_response('search_results.html',
                {'books': books, 'query': q})
    return render_to_response('search_form.html',
        {'error': error})
```

在改进后的视图中，若用户访问/search/并且没有带有GET数据，那么他将看到一个没有错误信息的表单；如果用户提交了一个空表单，那么它将看到错误提示信息，还有表单；最后，若用户提交了一个非空的值，那么他将看到搜索结果。

最后，我们再稍微改进一下这个表单，去掉冗余的部分。 既然已经将两个视图与URLs合并起来，/search/视图管理着表单的显示以及结果的显示，那么在search_form.html里表单的action值就没有必要硬编码的指定URL。 原先的代码是这样：

```
<form action="/search/" method="get">
```

现在改成这样：

```
<form action="" method="get">
```

action=""意味着表单将提交给与当前页面相同的URL。 这样修改之后，如果 `search()` 视图不指向其它页面的话，你将不必再修改 `action` 。

简单的验证

我们的搜索示例仍然相当地简单，特别从数据验证方面来讲；我们仅仅只验证搜索关键值是否为空。 然后许多HTML表单包含着比检测值是否为空更为复杂的验证。 我们都有在网站上见过类似以下的错误提示信息：

- 请输入一个有效的email地址， foo' 并不是一个有效的e-mail地址。
- 请输入5位数的U.S 邮政编码， 123并非是一个有效的邮政编码。
- 请输入YYYY-MM-DD格式的日期。
- 请输入8位数以上并至少包含一个数字的密码。

关于JavaScript验证

可以使用Javascript在客户端浏览器里对数据进行验证，这些知识已超出本书范围。 要注意：即使在客户端已经做了验证，但是服务器端仍必须再验证一次。 因为有些用户会将JavaScript关闭掉，并且还有一些怀有恶意的用户会尝试提交非法的数据来探测是否有可以攻击的机会。

除了在服务器端对用户提交的数据进行验证（例如在视图里验证），我们没有其他办法。JavaScript验证可以看作是额外的功能，但不能作为唯一的验证功能。

我们来调整一下search()视图，让她能够验证搜索关键词是否小于或等于20个字符。 （为来让例子更为显著，我们假设如果关键词超过20个字符将导致查询十分缓慢）。那么该如何实现呢？ 最简单的方式就是将逻辑处理直接嵌入到视图里，就像这样：

```
def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        **elif len(q) > 20:**
            **error = True**
        else:
            books = Book.objects.filter(title__icontains=q)
            return render_to_response('search_results.html',
                {'books': books, 'query': q})
    return render_to_response('search_form.html',
        {'error': error})
```

现在，如果尝试着提交一个超过20个字符的搜索关键词，系统不会执行搜索操作，而是显示一条错误提示信息。 但是，search_form.html里的这条提示信息是：”Please submit a search term.”，这显然是错误的， 所以我们需要更精确的提示信息：

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">Please submit a search term 20 characters or shorter.</p>
    {% endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

但像这样修改之后仍有一些问题。 我们包含万象的提示信息很容易使人产生困惑： 提交一个空表单怎么会出现一个关于20个字符限制的提示？ 所以，提示信息必须是详细的，明确的，不会产生疑议。

问题的实质在于我们只使用来一个布尔类型的变量来检测是否出错，而不是使用一个列表来记录相应的错误信息。 我们需要做如下的调整：

```
def search(request):
    **errors = []**
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            **errors.append('Enter a search term.')**
        elif len(q) > 20:
            **errors.append('Please enter at most 20 characters.')**
        else:
            books = Book.objects.filter(title__icontains=q)
            return render_to_response('search_results.html',
                {'books': books, 'query': q})
    return render_to_response('search_form.html',
        {**'errors': errors** })
```

接着，我们要修改一下search_form.html模板，现在需要显示一个errors列表而不是一个布尔判断。

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    **{% if errors %}**
        **<ul>**
            **{% for error in errors %}**
            **<li>{{ error }}</li>**
            **{% endfor %}**
        **</ul>**
    **{% endif %}**
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

编写Contact表单

虽然我们一直使用书籍搜索的示例表单，并将起改进的很完美，但是这还是相当的简陋：只包含一个字段，q。这简单的例子，我们不需要使用Django表单库来处理。但是复杂一点的表单就需要多方面的处理，我们现在来一下一个较为复杂的例子：站点联系表单。

这个表单包括用户提交的反馈信息，一个可选的e-mail回信地址。当这个表单提交并且数据通过验证后，系统将自动发送一封包含题用户提交的信息的e-mail给站点工作人员。

我们从contact_form.html模板入手：

```
<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if errors %}
        <ul>
            {% for error in errors %}
            <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}

    <form action="/contact/" method="post">
        <p>Subject: <input type="text" name="subject"></p>
        <p>Your e-mail (optional): <input type="text" name="email"></p>
        <p>Message: <textarea name="message" rows="10" cols="50"></textarea></p>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

我们定义了三个字段： 主题，e-mail和反馈信息。 除了e-mail字段为可选，其他两个字段都是必填项。 注意，这里我们使用method="post"而非method="get"，因为这个表单会有一个服务器端的操作：发送一封e-mail。 并且，我们复制了前一个模板search_form.html中错误信息显示的代码。

如果我们顺着上一节编写search()视图的思路，那么一个contact()视图代码应该像这样：

```
from django.core.mail import send_mail
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response

def contact(request):
    errors = []
    if request.method == 'POST':
        if not request.POST.get('subject', ''):
            errors.append('Enter a subject.')
        if not request.POST.get('message', ''):
            errors.append('Enter a message.')
        if request.POST.get('email') and '@' not in request.POST['email']:
            errors.append('Enter a valid e-mail address.')
        if not errors:
            send_mail(
                request.POST['subject'],
                request.POST['message'],
                request.POST.get('email', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    return render_to_response('contact_form.html',
        {'errors': errors})
```

（如果按照书中的示例做下来，这这里可能乎产生一个疑问：contact()视图是否要放在books/views.py这个文件里。 但是contact()视图与books应用没有任何关联，那么这个视图应该可以放在别的地方？ 这毫无紧要，只要在URLconf里正确设置URL与视图之间的映射，Django会正确处理的。 笔者个人喜欢创建一个contact的文件夹，与books文件夹同级。这个文件夹中包括空的**init**.py和views.py两个文件。

现在来分析一下以上的代码：

> 确认request.method的值是'POST'。用户浏览表单时这个值并不存在，当且仅当表单被提交时这个值才出现。 （在后面的例子中，request.method将会设置为'GET'，因为在普通的网页浏览中，浏览器都使用GET，而非POST）。判断request.method的值很好地帮助我们将表单显示与表单处理隔离开来。
>
> 我们使用request.POST代替request.GET来获取提交过来的数据。 这是必须的，因为contact_form.html里表单使用的是method="post"。如果在视图里通过POST获取数据，那么request.GET将为空。
>
> 这里，有两个必填项，subject 和 message，所以需要对这两个进行验证。 注意，我们使用request.POST.get()方法，并提供一个空的字符串作为默认值；这个方法很好的解决了键丢失与空数据问题。
>
> 虽然email非必填项，但如果有提交她的值则我们也需进行验证。 我们的验证算法相当的薄弱，仅验证值是否包含@字符。 在实际应用中，需要更为健壮的验证机制（Django提供这些验证机制，稍候我们就会看到）。
>
> 我们使用了django.core.mail.send_mail函数来发送e-mail。 这个函数有四个必选参数：主题，正文，寄信人和收件人列表。 send_mail是Django的EmailMessage类的一个方便的包装，EmailMessage类提供了更高级的方法，比如附件，多部分邮件，以及对于邮件头部的完整控制。
>
> 注意，若要使用send_mail()函数来发送邮件，那么服务器需要配置成能够对外发送邮件，并且在Django中设置出站服务器地址。 参见规范：http://docs.djangoproject.com/en/dev/topics/email/
>
> 当邮件发送成功之后，我们使用HttpResponseRedirect对象将网页重定向至一个包含成功信息的页面。 包含成功信息的页面这里留给读者去编写（很简单 一个视图/URL映射/一份模板即可），但是我们要解释一下为何重定向至新的页面，而不是在模板中直接调用render_to_response()来输出。
>
> 原因就是： 若用户刷新一个包含POST表单的页面，那么请求将会重新发送造成重复。 这通常会造成非期望的结果，比如说重复的数据库记录；在我们的例子中，将导致发送两封同样的邮件。 如果用户在POST表单之后被重定向至另外的页面，就不会造成重复的请求了。
>
> 我们应每次都给成功的POST请求做重定向。 这就是web开发的最佳实践。

contact()视图可以正常工作，但是她的验证功能有些复杂。 想象一下假如一个表单包含一打字段，我们真的将必须去编写每个域对应的if判断语句？

另外一个问题是表单的重新显示。若数据验证失败后，返回客户端的表单中各字段最好是填有原来提交的数据，以便用户查看哪里出现错误（用户也不需再次填写正确的字段值）。 我们可以手动地将原来的提交数据返回给模板，并且必须编辑HTML里的各字段来填充原来的值。

```
# views.py

def contact(request):
    errors = []
    if request.method == 'POST':
        if not request.POST.get('subject', ''):
            errors.append('Enter a subject.')
        if not request.POST.get('message', ''):
            errors.append('Enter a message.')
        if request.POST.get('email') and '@' not in request.POST['email']:
            errors.append('Enter a valid e-mail address.')
        if not errors:
            send_mail(
                request.POST['subject'],
                request.POST['message'],
                request.POST.get('email', `'noreply@example.com`_'),
                [`'siteowner@example.com`_'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    return render_to_response('contact_form.html', {
        'errors': errors,
        **'subject': request.POST.get('subject', ''),**
        **'message': request.POST.get('message', ''),**
        **'email': request.POST.get('email', ''),**
    })

# contact_form.html

<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if errors %}
        <ul>
            {% for error in errors %}
            <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}

    <form action="/contact/" method="post">
        <p>Subject: <input type="text" name="subject" **value="{{ subject }}"** ></p>
        <p>Your e-mail (optional): <input type="text" name="email" **value="{{ email }}"*
        <p>Message: <textarea name="message" rows="10" cols="50">**{{ message }}**</texta
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

这看起来杂乱，且写的时候容易出错。 希望你开始明白使用高级库的用意——负责处理表单及相关校验任务。

第一个Form类

Django带有一个form库，称为django.forms，这个库可以处理我们本章所提到的包括HTML表单显示以及验证。接下来我们来深入了解一下form库，并使用她来重写contact表单应用。

Django的newforms库

在Django社区上会经常看到django.newforms这个词语。当人们讨论django.newforms，其实就是我们本章里面介绍的django.forms。

改名其实有历史原因的。当Django一次向公众发行时，它有一个复杂难懂的表单系统：`django.forms`。后来它被完全重写了，新的版本改叫作：`django.newforms`，这样人们还可以通过名称，使用旧版本。当Django 1.0发布时，旧版本 `django.forms` 就不再使用了，而 `django.newforms` 也终于可以名正言顺的叫做：`django.forms`。

表单框架最主要的用法是，为每一个将要处理的HTML的 `<Form>` 定义一个 `Form` 类。在这个例子中，我们只有一个 `<Form>`，因此我们只需定义一个 `Form` 类。这个类可以存在于任何地方，甚至直接写在 `views.py` 文件里也行，但是社区的惯例是把 `Form` 类都放到一个文件中：`forms.py`。在存放 `views.py` 的目录中，创建这个文件，然后输入：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    email = forms.EmailField(required=False)
    message = forms.CharField()
```

这看上去简单易懂，并且很像在模块中使用的语法。表单中的每一个字段（域）作为 `Form` 类的属性，被展现成 `Field` 类。这里只用到 `CharField` 和 `EmailField` 类型。每一个字段都默认是必填。要使 `email` 成为可选项，我们需要指定 `required=False`。

让我们钻研到Python解释器里面看看这个类做了些什么。它做的第一件事是将自己显示成HTML：

```
>>> from contact.forms import ContactForm
>>> f = ContactForm()
>>> print f
<tr><th><label for="id_subject">Subject:</label></th><td><input type="text" name="subject
<tr><th><label for="id_email">Email:</label></th><td><input type="text" name="email" id="
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message
```

为了便于访问，Django用 `<label>` 标志，为每一个字段添加了标签。这个做法使默认行为尽可能合适。

默认输出按照HTML的< `table` >格式，另外有一些其它格式的输出：

```
>>> print f.as_ul()
<li><label for="id_subject">Subject:</label> <input type="text" name="subject" id="id_sub
<li><label for="id_email">Email:</label> <input type="text" name="email" id="id_email" />
<li><label for="id_message">Message:</label> <input type="text" name="message" id="id_mes
>>> print f.as_p()
<p><label for="id_subject">Subject:</label> <input type="text" name="subject" id="id_subj
<p><label for="id_email">Email:</label> <input type="text" name="email" id="id_email" /><
<p><label for="id_message">Message:</label> <input type="text" name="message" id="id_mess
```

请注意，标签<table>、<ul>、<form>的开闭合标记没有包含于输出当中，这样你就可以添加额外的行或者自定义格式。

这些类方法只是一般情况下用于快捷显示完整表单的方法。 你同样可以用HTML显示个别字段：

```
>>> print f['subject']
<input type="text" name="subject" id="id_subject" />
>>> print f['message']
<input type="text" name="message" id="id_message" />
```

`Form` 对象做的第二件事是校验数据。 为了校验数据，我们创建一个新的对 `Form` 象，并且传入一个与定义匹配的字典类型数据：

```
>>> f = ContactForm({'subject': 'Hello', 'email': 'adrian@example.com', 'message': 'Nice
```

一旦你对一个 `Form` 实体赋值，你就得到了一个绑定form：

```
>>> f.is_bound
True
```

调用任何绑定form的 `is_valid()` 方法，就可以知道它的数据是否合法。 我们已经为每个字段传入了值，因此整个 `Form` 是合法的：

```
>>> f.is_valid()
True
```

如果我们不传入 `email` 值，它依然是合法的。因为我们指定这个字段的属性 `required=False` ：

```
>>> f = ContactForm({'subject': 'Hello', 'message': 'Nice site!'})
>>> f.is_valid()
True
```

但是，如果留空 `subject` 或 `message` ，整个 `Form` 就不再合法了：

```
>>> f = ContactForm({'subject': 'Hello'})
>>> f.is_valid()
False
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f.is_valid()
False
```

你可以逐一查看每个字段的出错消息：

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f['message'].errors
[u'This field is required.']
>>> f['subject'].errors
[]
>>> f['email'].errors
[]
```

每一个邦定 `Form` 实体都有一个 `errors` 属性，它为你提供了一个字段与错误消息相映射的字典表。

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f.errors
{'message': [u'This field is required.']}
```

最终，如果一个 `Form` 实体的数据是合法的，它就会有一个可用的 `cleaned_data` 属性。 这是一个包含干净的提交数据的字典。 Django的form框架不但校验数据，它还会把它们转换成相应的Python类型数据，这叫做清理数据。

```
>>> f = ContactForm({subject': Hello, email: adrian@example.com, message: Nice site!})
>>> f.is_valid()
True
>>> f.cleaned_data
{message': uNice site!, email: uadrian@example.com, subject: uHello}
```

我们的contact form只涉及字符串类型，它们会被清理成Unicode对象。如果我们使用整数型或日期型，form框架会确保方法使用合适的Python整数型或 `datetime.date` 型对象。

在视图中使用Form对象

在学习了关于 `Form` 类的基本知识后，你会看到我们如何把它用到视图中，取代 `contact()` 代码中不整齐的部分。 一下示例说明了我们如何用forms框架重写 `contact()` ：

```
# views.py

from django.shortcuts import render_to_response
from mysite.contact.forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('email', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render_to_response('contact_form.html', {'form': form})

# contact_form.html

<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if form.errors %}
        <p style="color: red;">
            Please correct the error{{ form.errors|pluralize }} below.
        </p>
    {% endif %}

    <form action="" method="post">
        <table>
            {{ form.as_table }}
        </table>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

看看，我们能移除这么多不整齐的代码！ Django的forms框架处理HTML显示、数据校验、数据清理和表单错误重现。

尝试在本地运行。 装载表单，先留空所有字段提交空表单；继而填写一个错误的邮箱地址再尝试提交表单；最后再用正确数据提交表单。 （根据服务器的设置，当 `send_mail()` 被调用时，你将得到一个错误提示。而这是另一个问题。）

改变字段显示

你可能首先注意到：当你在本地显示这个表单的时，`message` 字段被显示成 `input type=”text”` ，而它应该被显示成< `textarea` >。我们可以通过设置 *widget* 来修改它：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    email = forms.EmailField(required=False)
    message = forms.CharField(**widget=forms.Textarea** )
```

forms框架把每一个字段的显示逻辑分离到一组部件（widget）中。 每一个字段类型都拥有一个默认的部件，我们也可以容易地替换掉默认的部件，或者提供一个自定义的部件。

考虑一下 `Field` 类表现 校验逻辑 ，而部件表现 显示逻辑 。

设置最大长度

一个最经常使用的校验要求是检查字段长度。 另外，我们应该改进 `ContactForm` ，使 `subject` 限制在100个字符以内。 为此，仅需为 `CharField` 提供 `max_length` 参数，像这样：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(**max_length=100** )
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)
```

选项 `min_length` 参数同样可用。

设置初始值

让我们再改进一下这个表单：为字 `subject` 段添加 初始值 ： `"I love your site!"` （一点建议，但没坏处。）为此，我们可以在创建 `Form` 实体时，使用 `initial` 参数：

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('email', `'noreply@example.com`_'),
                [`'siteowner@example.com`_'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm(
            **initial={'subject': 'I love your site!'}**
        )
    return render_to_response('contact_form.html', {'form': form})
```

现在， `subject` 字段将被那个句子填充。

请注意，传入 初始值 数据和传入数据以 绑定 表单是有区别的。 最大的区别是，如果仅传入初始值 数据，表单是*unbound*的，那意味着它没有错误消息。

自定义校验规则

假设我们已经发布了反馈页面了，email已经开始源源不断地涌入了。 这里有一个问题： 一些提交的消息只有一两个字，我们无法得知详细的信息。 所以我们决定增加一条新的校验： 来点专业精神，最起码写四个字，拜托。

我们有很多的方法把我们的自定义校验挂在Django的form上。 如果我们的规则会被一次又一次的使用，我们可以创建一个自定义的字段类型。 大多数的自定义校验都是一次性的，可以直接绑定到form类.

我们希望 `message` 字段有一个额外的校验，我们增加一个 `clean_message()` 方法到 `Form` 类：

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)

    def clean_message(self):
        message = self.cleaned_data['message']
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

Django的form系统自动寻找匹配的函数方法，该方法名称以 `clean_` 开头，并以字段名称结束。 如果有这样的方法，它将在校验时被调用。

特别地， `clean_message()` 方法将在指定字段的默认校验逻辑执行 之后 被调用。（本例中，在必填 `CharField` 这个校验逻辑之后。）因为字段数据已经被部分处理，所以它被从 `self.cleaned_data` 中提取出来了。同样，我们不必担心数据是否为空，因为它已经被校验过了。

我们简单地使用了len()和split()的组合来计算单词的数量。 如果用户输入字数不足，我们抛出一个 `forms.ValidationError` 型异常。这个异常的描述会被作为错误列表中的一项显示给用户。

在函数的末尾显式地返回字段的值非常重要。 我们可以在我们自定义的校验方法中修改它的值（或者把它转换成另一种Python类型）。 如果我们忘记了这一步，None值就会返回，原始的数据就丢失掉了。

指定标签

HTML表单中自动生成的标签默认是按照规则生成的：用空格代替下划线，首字母大写。如 `email` 的标签是 `"Email"` 。（好像在哪听到过？ 是的，同样的逻辑被用于模块（model）中字段的 `verbose_name` 值。 我们在第五章谈到过。）

像在模块中做过的那样，我们同样可以自定义字段的标签。 仅需使用 `label` ，像这样：

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False, **label='Your e-mail address'** )
    message = forms.CharField(widget=forms.Textarea)
```

定制Form设计

在上面的 `contact_form.html` 模板中我们使用 `{{form.as_table}}` 显示表单，不过我们可以使用其他更精确控制表单显示的方法。

修改form的显示的最快捷的方式是使用CSS。 尤其是错误列表，可以增强视觉效果。自动生成的错误列表精确的使用 `<ul class=”errorlist”>` ，这样，我们就可以针对它们使用CSS。 下面的CSS让错误更加醒目了：

```
<style type="text/css">
    ul.errorlist {
        margin: 0;
        padding: 0;
    }
    .errorlist li {
        background-color: red;
        color: white;
        display: block;
        font-size: 10px;
        margin: 0 0 3px;
        padding: 4px 5px;
    }
</style>
```

虽然，自动生成HTML是很方便的，但是在某些时候，你会想覆盖默认的显示。{{form.as_table}}和其它的方法在开发的时候是一个快捷的方式，form的显示方式也可以在form中被方便地重写。

每一个字段部件(<input type=”text”>, <select>, <textarea>, 或者类似)都可以通过访问{{form.字段名}}进行单独的渲染。

```
<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if form.errors %}
        <p style="color: red;">
            Please correct the error{{ form.errors|pluralize }} below.
        </p>
    {% endif %}

    <form action="" method="post">
        <div class="field">
            {{ form.subject.errors }}
            <label for="id_subject">Subject:</label>
            {{ form.subject }}
        </div>
        <div class="field">
            {{ form.email.errors }}
            <label for="id_email">Your e-mail address:</label>
            {{ form.email }}
        </div>
        <div class="field">
            {{ form.message.errors }}
            <label for="id_message">Message:</label>
            {{ form.message }}
        </div>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

`{{ form.message.errors }}` 会在 <ul class="errorlist"> 里面显示，如果字段是合法的，或者form没有被绑定，就显示一个空字符串。 我们还可以把 `form.message.errors` 当作一个布尔值或者当它是list在上面做迭代， 例如：

```
<div class="field{% if form.message.errors %} errors{% endif %}">
    {% if form.message.errors %}
        <ul>
        {% for error in form.message.errors %}
            <li><strong>{{ error }}</strong></li>
        {% endfor %}
        </ul>
    {% endif %}
    <label for="id_message">Message:</label>
    {{ form.message }}
</div>
```

在校验失败的情况下, 这段代码会在包含错误字段的div的class属性中增加一个”errors”，在一个有序列表中显示错误信息。

下一章

这一章总结了本书的介绍材料，即所谓“核心教程”。 后面部分，从第八章到第十二章，将详细讲述高级（进阶）使用，包括如何配置一个Django应用程序（第十二章）。

在学习本书的前面七章后，我们终于对于使用Django构建自己的网站已经知道的够多了， 本书中剩余的材料将在你需要的时候帮助你补遗。

第八章我们将回头、并深入地讲解 视图和URLconfs（第三章已简单介绍）。

第八章：高级视图和URL配置

在第三章，我们已经对基本的Django视图和URL配置做了介绍。 在这一章，将进一步说明框架中这两个部分的高级机能。

URLconf 技巧

URLconf没什么特别的，就象 Django 中其它东西一样，它们只是 Python 代码。 你可以在几方面从中得到好处，正如下面所描述的。

流线型化(Streamlining)函数导入

看下这个 URLconf，它是建立在第三章的例子上:

```
from django.conf.urls.defaults import *
from mysite.views import hello, current_datetime, hours_ahead

urlpatterns = patterns('',
    (r'^hello/$', hello),
    (r'^time/$', current_datetime),
    (r'^time/plus/(\d{1,2})/$', hours_ahead),
)
```

正如第三章中所解释的，在 URLconf 中的每一个入口包括了它所关联的视图函数，直接传入了一个函数对象。 这就意味着需要在模块开始处导入视图函数。

但随着 Django 应用变得复杂，它的 URLconf 也在增长，并且维护这些导入可能使得管理变麻烦。 (对每个新的view函数，你不得不记住要导入它，并且采用这种方法会使导入语句将变得相当长。)可以通过导入 views 模块本身来避免这个麻烦。 下面例子的URLconf与前一个等价：

```
from django.conf.urls.defaults import *
**from mysite import views**

urlpatterns = patterns('',
    (r'^hello/$', **views.hello** ),
    (r'^time/$', **views.current_datetime** ),
    (r'^time/plus/(d{1,2})/$', **views.hours_ahead** ),
)
```

Django 还提供了另一种方法可以在 URLconf 中为某个特别的模式指定视图函数： 你可以传入一个包含模块名和函数名的字符串，而不是函数对象本身。 继续示例:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^hello/$', **'mysite.views.hello'** ),
    (r'^time/$', **'mysite.views.current_datetime'** ),
    (r'^time/plus/(d{1,2})/$', **'mysite.views.hours_ahead'** ),
)
```

(注意视图名前后的引号。 应该使用带引号的 `'mysite.views.current_datetime'` 而不是 `mysite.views.current_datetime` 。)

使用这个技术，就不必导入视图函数了；Django 会在第一次需要它时根据字符串所描述的视图函数的名字和路径，导入合适的视图函数。

当使用字符串技术时，你可以采用更简化的方式：提取出一个公共视图前缀。 在我们的URLconf例子中，每个视图字符串的开始部分都是```\，造成重复输入。 我们可以把公共的前缀提取出来，作为第一个参数传给\ ```函数：

System Message: WARNING/2 (<string> , line 99); *backlink*

Inline literal start-string without end-string.

```
from django.conf.urls.defaults import *

urlpatterns = patterns(**'mysite.views'** ,
    (r'^hello/$', **'hello'** ),
    (r'^time/$', **'current_datetime'** ),
    (r'^time/plus/(d{1,2})/$', **'hours_ahead'** ),
)
```

注意既不要在前缀后面跟着一个点号(`"."`)，也不要在视图字符串前面放一个点号。 Django 会自动处理它们。

牢记这两种方法，哪种更好一些呢？ 这取决于你的个人编码习惯和需要。

字符串方法的好处如下:

- 更紧凑，因为不需要你导入视图函数。
- 如果你的视图函数存在于几个不同的 Python 模块的话，它可以使得 URLconf 更易读和管理。

函数对象方法的好处如下:

- 更容易对视图函数进行包装(wrap)。 参见本章后面的《包装视图函数》一节。
- 更 Pythonic，就是说，更符合 Python 的传统，如把函数当成对象传递。

两个方法都是有效的，甚至你可以在同一个 URLconf 中混用它们。 决定权在你。

使用多个视图前缀

在实践中，如果你使用字符串技术，特别是当你的 URLconf 中没有一个公共前缀时，你最终可能混合视图。 然而，你仍然可以利用视图前缀的简便方式来减少重复。 只要增加多个 `patterns()` 对象，象这样:

旧的:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^hello/$', 'mysite.views.hello'),
    (r'^time/$', 'mysite.views.current_datetime'),
    (r'^time/plus/(\d{1,2})/$', 'mysite.views.hours_ahead'),
    (r'^tag/(\w+)/$', 'weblog.views.tag'),
)
```

新的:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.views',
    (r'^hello/$', 'hello'),
    (r'^time/$', 'current_datetime'),
    (r'^time/plus/(\d{1,2})/$', 'hours_ahead'),
)

urlpatterns += patterns('weblog.views',
    (r'^tag/(\w+)/$', 'tag'),
)
```

整个框架关注的是存在一个名为 `urlpatterns` 的模块级别的变量。如上例，这个变量可以动态生成。 这里我们要特别说明一下，patterns()返回的对象是可相加的，这个特性可能是大家没有想到的。

调试模式中的特例

说到动态构建 `urlpatterns` ，你可能想利用这一技术，在 Django 的调试模式下修改 URLconf 的行为。 为了做到这一点，只要在运行时检查 `DEBUG` 配置项的值即可，如:

```
from django.conf import settings
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^$', views.homepage),
    (r'^(\d{4})/([a-z]{3})/$', views.archive_month),
)

if settings.DEBUG:
    urlpatterns += patterns('',
        (r'^debuginfo/$', views.debug),
    )
```

在这个例子中，URL链接 `/debuginfo/` 只在你的 `DEBUG` 配置项设为 `True` 时才有效。

使用命名组

在目前为止的所有 URLconf 例子中，我们使用简单的无命名 正则表达式组，即，在我们想要捕获的URL部分上加上小括号，Django 会将捕获的文本作为位置参数传递给视图函数。 在更高级的用法中，还可以使用 命名 正则表达式组来捕获URL，并且将其作为 关键字 参数传给视图。

关键字参数 对比 位置参数

一个 Python 函数可以使用关键字参数或位置参数来调用，在某些情况下，可以同时进行使用。 在关键字参数调用中，你要指定参数的名字和传入的值。 在位置参数调用中，你只需传入参数，不需要明确指明哪个参数与哪个值对应，它们的对应关系隐含在参数的顺序中。

例如，考虑这个简单的函数:

```
def sell(item, price, quantity):
    print "Selling %s unit(s) of %s at %s" % (quantity, item, price)
```

为了使用位置参数来调用它，你要按照在函数定义中的顺序来指定参数。

```
sell('Socks', '$2.50', 6)
```

为了使用关键字参数来调用它，你要指定参数名和值。 下面的语句是等价的:

```
sell(item='Socks', price='$2.50', quantity=6)
sell(item='Socks', quantity=6, price='$2.50')
sell(price='$2.50', item='Socks', quantity=6)
sell(price='$2.50', quantity=6, item='Socks')
sell(quantity=6, item='Socks', price='$2.50')
sell(quantity=6, price='$2.50', item='Socks')
```

最后，你可以混合关键字和位置参数，只要所有的位置参数列在关键字参数之前。 下面的语句与前面的例子是等价:

```
sell('Socks', '$2.50', quantity=6)
sell('Socks', price='$2.50', quantity=6)
sell('Socks', quantity=6, price='$2.50')
```

在 Python 正则表达式中，命名的正则表达式组的语法是 `(?P<name>pattern)` ，这里 `name` 是组的名字，而 `pattern` 是匹配的某个模式。

下面是一个使用无名组的 URLconf 的例子:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(\d{4})/$', views.year_archive),
    (r'^articles/(\d{4})/(\d{2})/$', views.month_archive),
)
```

下面是相同的 URLconf，使用命名组进行了重写:

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(?P<year>\d{4})/$', views.year_archive),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', views.month_archive),
)
```

这段代码和前面的功能完全一样，只有一个细微的差别： 取的值是以关键字参数的方式而不是以位置参数的方式传递给视图函数的。

例如，如果不带命名组，请求 `/articles/2006/03/` 将会等同于这样的函数调用：

```
month_archive(request, '2006', '03')
```

而带命名组，同样的请求就会变成这样的函数调用：

```
month_archive(request, year='2006', month='03')
```

使用命名组可以让你的URLconfs更加清晰，减少搞混参数次序的潜在BUG，还可以让你在函数定义中对参数重新排序。 接着上面这个例子，如果我们想修改URL把月份放到 年份的 前面，而不使用命名组的话，我们就不得不去修改视图 `month_archive` 的参数次序。 如果我们使用命名组的话，修改URL里提取参数的次序对视图没有影响。

当然，命名组的代价就是失去了简洁性： 一些开发者觉得命名组的语法丑陋和显得冗余。 命名组的另一个好处就是可读性强。

理解匹配/分组算法

需要注意的是如果在URLconf中使用命名组，那么命名组和非命名组是不能同时存在于同一个URLconf的模式中的。 如果你这样做，Django不会抛出任何错误，但你可能会发现你的URL并没有像你预想的那样匹配正确。 具体地，以下是URLconf解释器有关正则表达式中命名组和 非命名组所遵循的算法:

- 如果有任何命名的组，Django会忽略非命名组而直接使用命名组。
- 否则，Django会把所有非命名组以位置参数的形式传递。

- 在以上的两种情况，Django同时会以关键字参数的方式传递一些额外参数。 更具体的信息可参考下一节。

传递额外的参数到视图函数中

有时你会发现你写的视图函数是十分类似的，只有一点点的不同。 比如说，你有两个视图，它们的内容是一致的，除了它们所用的模板不太一样：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^foo/$', views.foo_view),
    (r'^bar/$', views.bar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foo_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template1.html', {'m_list': m_list})

def bar_view(request):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response('template2.html', {'m_list': m_list})
```

我们在这代码里面做了重复的工作，不够简练。 起初你可能会想，通过对两个URL都使用同样的视图，在URL中使用括号捕捉请求，然后在视图中检查并决定使用哪个模板来去除代码的冗余，就像这样：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^(foo)/$', views.foobar_view),
    (r'^(bar)/$', views.foobar_view),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, url):
    m_list = MyModel.objects.filter(is_new=True)
    if url == 'foo':
        template_name = 'template1.html'
    elif url == 'bar':
        template_name = 'template2.html'
    return render_to_response(template_name, {'m_list': m_list})
```

这种解决方案的问题还是老缺点，就是把你的URL耦合进你的代码里面了。 如果你打算把 `/foo/` 改成 `/fooey/` 的话，那么你就得记住要去改变视图里面的代码。

对一个可选URL配置参数的优雅解决方法： URLconf里面的每一个模式都可以包含第三个数据： 一个关键字参数的字典：

有了这个概念以后，我们就可以把我们现在的例子改写成这样：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^foo/$', views.foobar_view, {'template_name': 'template1.html'}),
    (r'^bar/$', views.foobar_view, {'template_name': 'template2.html'}),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import MyModel

def foobar_view(request, template_name):
    m_list = MyModel.objects.filter(is_new=True)
    return render_to_response(template_name, {'m_list': m_list})
```

如你所见，这个例子中，URLconf指定了 `template_name` 。 而视图函数会把它当成另一个参数。

这种使用额外的URLconf参数的技术以最小的代价给你提供了向视图函数传递额外信息的一个好方法。 正因如此，这技术已被很多Django的捆绑应用使用，其中以我们将在第11章讨论的通用视图系统最为明显。

下面的几节里面有一些关于你可以怎样把额外URLconf参数技术应用到你自己的工程的建议。

伪造捕捉到的**URLconf**值

比如说你有匹配某个模式的一堆视图，以及一个并不匹配这个模式但视图逻辑是一样的URL。 这种情况下，你可以通过向同一个视图传递额外URLconf参数来伪造URL值的捕捉。

例如，你可能有一个显示某一个特定日子的某些数据的应用，URL类似这样的：

```
/mydata/jan/01/
/mydata/jan/02/
/mydata/jan/03/
# ...
/mydata/dec/30/
/mydata/dec/31/
```

这太简单了，你可以在一个URLconf中捕捉这些值，像这样（使用命名组的方法）：

```
urlpatterns = patterns('',
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

然后视图函数的原型看起来会是：

```
def my_view(request, month, day):
    # ....
```

这种解决方案很直接，没有用到什么你没见过的技术。 当你想添加另外一个使用 `my_view` 视图但不包含 `month` 和/或者 `day` 的URL时，问题就出现了。

比如你可能会想增加这样一个URL， `/mydata/birthday/` ， 这个URL等价于 `/mydata/jan/06/` 。这时你可以这样利用额外URLconf参数：

```
urlpatterns = patterns('',
    (r'^mydata/birthday/$', views.my_view, {'month': 'jan', 'day': '06'}),
    (r'^mydata/(?P<month>\w{3})/(?P<day>\d\d)/$', views.my_view),
)
```

在这里最帅的地方莫过于你根本不用改变你的视图函数。 视图函数只会关心它 获得 了 参数，它不会去管这些参数到底是捕捉回来的还是被额外提供的。month和day

创建一个通用视图

抽取出我们代码中共性的东西是一个很好的编程习惯。 比如，像以下的两个Python函数：

```
def say_hello(person_name):
    print 'Hello, %s' % person_name

def say_goodbye(person_name):
    print 'Goodbye, %s' % person_name
```

我们可以把问候语提取出来变成一个参数：

```
def greet(person_name, greeting):
    print '%s, %s' % (greeting, person_name)
```

通过使用额外的URLconf参数，你可以把同样的思想应用到Django的视图中。

了解这个以后，你可以开始创作高抽象的视图。 更具体地说，比如这个视图显示一系列的 `Event` 对象，那个视图显示一系列的 `BlogEntry` 对象，并意识到它们都是一个用来显示一系列对象的视图的特例，而对象的类型其实就是一个变量。

以这段代码作为例子：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^events/$', views.event_list),
    (r'^blog/entries/$', views.entry_list),
)

# views.py

from django.shortcuts import render_to_response
from mysite.models import Event, BlogEntry

def event_list(request):
    obj_list = Event.objects.all()
    return render_to_response('mysite/event_list.html', {'event_list': obj_list})

def entry_list(request):
    obj_list = BlogEntry.objects.all()
    return render_to_response('mysite/blogentry_list.html', {'entry_list': obj_list})
```

这两个视图做的事情实质上是一样的： 显示一系列的对象。 让我们把它们显示的对象的类型抽象出来：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import models, views

urlpatterns = patterns('',
    (r'^events/$', views.object_list, {'model': models.Event}),
    (r'^blog/entries/$', views.object_list, {'model': models.BlogEntry}),
)

# views.py

from django.shortcuts import render_to_response

def object_list(request, model):
    obj_list = model.objects.all()
    template_name = 'mysite/%s_list.html' % model.__name__.lower()
    return render_to_response(template_name, {'object_list': obj_list})
```

就这样小小的改动，我们突然发现我们有了一个可复用的，模型无关的视图！ 从现在开始，当我们需要一个视图来显示一系列的对象时，我们可以简简单单的重用这一个 `object_list` 视图，而无须另外写视图代码了。 以下是我们做过的事情：

- 我们通过 `model` 参数直接传递了模型类。 额外URLconf参数的字典是可以传递任何类型的对象，而不仅仅只是字符串。
- 这一行： `model.objects.all()` 是 鸭子界定 （原文：
- 我们使用 `model.__name__.lower()` 来决定模板的名字。 每个Python的类都有一个 `__name__` 属性返回类名。 这特性在当我们直到运行时刻才知道对象类型的这种情况下很有用。 比如， `BlogEntry` 类的 `__name__` 就是字符串 `'BlogEntry'` 。

- 这个例子与前面的例子稍有不同，我们传递了一个通用的变量名给模板。 当然我们可以轻易的把这个变量名改成 `blogentry_list` 或者 `event_list` ，不过我们打算把这当作练习留给读者。

因为数据库驱动的网站都有一些通用的模式，Django提供了一个通用视图的集合，使用它可以节省你的时间。 我们将会在下一章讲讲Django的内置通用视图。

提供视图配置选项

如果你发布一个Django的应用，你的用户可能会希望配置上能有些自由度。 这种情况下，为你认为用户可能希望改变的配置选项添加一些钩子到你的视图中会是一个很好的主意。 你可以用额外URLconf参数实现。

一个应用中比较常见的可供配置代码是模板名字：

```
def my_view(request, template_name):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

了解捕捉值和额外参数之间的优先级 额外的选项

当冲突出现的时候，额外URLconf参数优先于捕捉值。 也就是说，如果URLconf捕捉到的一个命名组变量和一个额外URLconf参数包含的变量同名时，额外URLconf参数的值会被使用。

例如，下面这个URLconf：

```
from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^mydata/(?P<id>\d+)/$', views.my_view, {'id': 3}),
)
```

这里，正则表达式和额外字典都包含了一个 `id` 。硬编码的（额外字典的） `id` 将优先使用。 就是说任何请求（比如， `/mydata/2/` 或者 `/mydata/432432/` ）都会作 `id` 设置为 `3` 对待，不管URL里面能捕捉到什么样的值。

聪明的读者会发现在这种情况下，在正则表达式里面写上捕捉是浪费时间的，因为 `id` 的值总是会被字典中的值覆盖。 没错，我们说这个的目的只是为了让你不要犯这样的错误。

使用缺省视图参数

另外一个方便的特性是你可以给一个视图指定默认的参数。 这样，当没有给这个参数赋值的时候将会使用默认的值。

例子：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^blog/$', views.page),
    (r'^blog/page(?P<num>\d+)/$', views.page),
)

# views.py

def page(request, num='1'):
    # Output the appropriate page of blog entries, according to num.
    # ...
```

在这里，两个URL表达式都指向了同一个视图 `views.page` ，但是第一个表达式没有传递任何参数。 如果匹配到了第一个样式， `page()` 函数将会对参数 `num` 使用默认值 `"1"` ，如果第二个表达式匹配成功， `page()` 函数将使用正则表达式传递过来的num的值。

（注：我们已经注意到设置默认参数值是字符串 `‘1’` ，不是整数 `1` 。为了保持一致，因为捕捉给 `num` 的值总是字符串。

就像前面解释的一样，这种技术与配置选项的联用是很普遍的。 以下这个例子比提供视图配置选项一节中的例子有些许的改进。

```
def my_view(request, template_name='mysite/my_view.html'):
    var = do_something()
    return render_to_response(template_name, {'var': var})
```

特殊情况下的视图

有时你有一个模式来处理在你的URLconf中的一系列URL，但是有时候需要特别处理其中的某个URL。 在这种情况下，要使用将URLconf中把特殊情况放在首位的线性处理方式 。

比方说，你可以考虑通过下面这个URLpattern所描述的方式来向Django的管理站点添加一个目标页面

```
urlpatterns = patterns('',
    # ...
    ('^([^/]+)/([^/]+)/add/$', views.add_stage),
    # ...
)
```

这将匹配像 `/myblog/entries/add/` 和 `/auth/groups/add/` 这样的URL 。然而，对于用户对象的添加页面（ `/auth/user/add/` ）是个特殊情况，因为它不会显示所有的表单域，它显示两个密码域等等。 我们 可以 在视图中特别指出以解决这种情况：

```
def add_stage(request, app_label, model_name):
    if app_label == 'auth' and model_name == 'user':
        # do special-case code
    else:
        # do normal code
```

不过，就如我们多次在这章提到的，这样做并不优雅：因为它把URL逻辑放在了视图中。更优雅的解决方法是，我们要利用URLconf从顶向下的解析顺序这个特点：

```
urlpatterns = patterns('',
    # ...
    ('^auth/user/add/$', views.user_add_stage),
    ('^([^/]+)/([^/]+)/add/$', views.add_stage),
    # ...
)
```

在这种情况下，象 `/auth/user/add/` 的请求将会被 `user_add_stage` 视图处理。尽管URL也匹配第二种模式，它会先匹配上面的模式。（这是短路逻辑。）

从URL中捕获文本

每个被捕获的参数将被作为纯Python字符串来发送，而不管正则表达式中的格式。举个例子，在这行URLConf中：

```
(r'^articles/(?P<year>\d{4})/$', views.year_archive),
```

尽管 `\d{4}` 将只匹配整数的字符串，但是参数 `year` 是作为字符串传至 `views.year_archive()` 的，而不是整型。

当你在写视图代码时记住这点很重要，许多Python内建的方法对于接受的对象的类型很讲究。许多内置Python函数是挑剔的（这是理所当然的）只接受特定类型的对象。一个典型的的错误就是用字符串值而不是整数值来创建 `datetime.date` 对象：

```
>>> import datetime
>>> datetime.date('1993', '7', '9')
Traceback (most recent call last):
    ...
TypeError: an integer is required
>>> datetime.date(1993, 7, 9)
datetime.date(1993, 7, 9)
```

回到URLconf和视图处，错误看起来很可能是这样：

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    (r'^articles/(\d{4})/(\d{2})/(\d{2})/$', views.day_archive),
)

# views.py

import datetime

def day_archive(request, year, month, day):
    # The following statement raises a TypeError!
    date = datetime.date(year, month, day)
```

因此， `day_archive()` 应该这样写才是正确的：

```
def day_archive(request, year, month, day):
    date = datetime.date(int(year), int(month), int(day))
```

注意，当你传递了一个并不完全包含数字的字符串时， `int()` 会抛出 `ValueError` 的异常，不过我们已经避免了这个错误，因为在URLconf的正则表达式中已经确保只有包含数字的字符串才会传到这个视图函数中。

决定URLconf搜索的东西

当一个请求进来时，Django试着将请求的URL作为一个普通Python字符串进行URLconf模式匹配（而不是作为一个Unicode字符串）。 这并不包括 `GET` 或 `POST` 参数或域名。 它也不包括第一个斜杠，因为每个URL必定有一个斜杠。

例如，在向 `http://www.example.com/myapp/` 的请求中，Django将试着去匹配 `myapp/` 。在向 `http://www.example.com/myapp/?page=3` 的请求中，Django同样会去匹配 `myapp/` 。

在解析URLconf时，请求方法（例如， `POST` ， `GET` ， `HEAD` ）并 不会 被考虑。 换而言之，对于相同的URL的所有请求方法将被导向到相同的函数中。 因此根据请求方法来处理分支是视图函数的责任。

视图函数的高级概念

说到关于请求方法的分支，让我们来看一下可以用什么好的方法来实现它。 考虑这个URLconf/view 设计:

```
# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    # ...
    (r'^somepage/$', views.some_page),
    # ...
)

# views.py

from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render_to_response

def some_page(request):
    if request.method == 'POST':
        do_something_for_post()
        return HttpResponseRedirect('/someurl/')
    elif request.method == 'GET':
        do_something_for_get()
        return render_to_response('page.html')
    else:
        raise Http404()
```

在这个示例中， `some_page()` 视图函数对 `POST` 和 `GET` 这两种请求方法的处理完全不同。 它们唯一的共同点是共享一个URL地址： `/somepage/.` 正如大家所看到的，在同一个视图函数中对 `POST` 和 `GET` 进行处理是一种很初级也很粗糙的做法。 一个比较好的设计习惯应该是，用两个分开的视图函数——一个处理 `POST` 请求，另一个处理 `GET` 请求，然后在相应的地方分别进行调用。

我们可以像这样做：先写一个视图函数然后由它来具体分派其它的视图，在之前或之后可以执行一些我们自定的程序逻辑。 下边的示例展示了这个技术是如何帮我们改进前边那个简单的 `some_page()` 视图的：

```
# views.py

from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render_to_response

def method_splitter(request, GET=None, POST=None):
    if request.method == 'GET' and GET is not None:
        return GET(request)
    elif request.method == 'POST' and POST is not None:
        return POST(request)
    raise Http404

def some_page_get(request):
    assert request.method == 'GET'
    do_something_for_get()
    return render_to_response('page.html')

def some_page_post(request):
    assert request.method == 'POST'
    do_something_for_post()
    return HttpResponseRedirect('/someurl/')

# urls.py

from django.conf.urls.defaults import *
from mysite import views

urlpatterns = patterns('',
    # ...
    (r'^somepage/$', views.method_splitter, {'GET': views.some_page_get, 'POST': views.so
    # ...
)
```

让我们从头看一下代码是如何工作的：

> 我们写了一个新的视图， `method_splitter()` ，它根据 `request.method` 返回的值来调用相应的视图。可以看到它带有两个关键参数， `GET` 和 `POST` ，也许应该是 视图函数 。如果 `request.method` 返回 `GET` ，那它就会自动调用 `GET` 视图。 如果 `request.method` 返回的是 `POST` ，那它调用的就是 `POST` 视图。 如果 `request.method` 返回的是其它值（如： `HEAD` ），或者是没有把 `GET` 或 `POST` 提交给此函数，那它就会抛出一个 `Http404` 错误。
>
> 在URLconf中，我们把 `/somepage/` 指到 `method_splitter()` 函数，并把视图函数额外需要用到的 `GET` 和 `POST` 参数传递给它。
>
> 最终，我们把 `some_page()` 视图分解到两个视图函数中 `some_page_get()` 和 `some_page_post()` 。这比把所有逻辑都挤到一个单一视图的做法要优雅得多。
>
> 注意，在技术上这些视图函数就不用再去检查 `request.method` 了，因为 `method_splitter()` 已经替它们做了。 （比如， `some_page_post()` 被调用的时候，我们可以确信 `request.method` 返回的值是 `post` 。）当然，这样做不止更安全也能更好的将代码文档化，这里我们做了一个假定，就是 `request.method` 能象我们所期望的那样工作。

现在我们就拥有了一个不错的，可以通用的视图函数了，里边封装着由 `request.method` 的返回值来分派不同的视图的程序。关于 `method_splitter()` 就不说什么了，当然，我们可以把它们重用到其它项目中。

然而，当我们做到这一步时，我们仍然可以改进 `method_splitter` 。从代码我们可以看到，它假设 `Get` 和 `POST` 视图除了 `request` 之外不需要任何其他的参数。那么，假如我们想要使用 `method_splitter` 与那种会从URL里捕捉字符，或者会接收一些可选参数的视图一起工作时该怎么办呢？

为了实现这个，我们可以使用Python中一个优雅的特性 带星号的可变参数 我们先展示这些例子，接着再进行解释

```
def method_splitter(request, *args, **kwargs):
    get_view = kwargs.pop('GET', None)
    post_view = kwargs.pop('POST', None)
    if request.method == 'GET' and get_view is not None:
        return get_view(request, *args, **kwargs)
    elif request.method == 'POST' and post_view is not None:
        return post_view(request, *args, **kwargs)
    raise Http404
```

这里,我们重构method_splitter(),去掉了GET和POST两个关键字参数,改而支持使用*args*和和**kwargs(*注意号) 这是一个Python特性，允许函数接受动态的、可变数量的、参数名只在运行时可知的参数。 如果你在函数定义时,只在参数前面加一个号,所有传递给函数的参数将会保存为一个元组. 如果你在函数定义时,在参数前面加两个号,所有传递给函数的关键字参数,将会保存为一个字典

例如，对于这个函数

```
def foo(*args, **kwargs):
    print "Positional arguments are:"
    print args
    print "Keyword arguments are:"
    print kwargs
```

看一下它是怎么工作的

```
>>> foo(1, 2, 3)
Positional arguments are:
(1, 2, 3)
Keyword arguments are:
{}
>>> foo(1, 2, name='Adrian', framework='Django')
Positional arguments are:
(1, 2)
Keyword arguments are:
{'framework': 'Django', 'name': 'Adrian'}
```

回过头来看，你能发现我们用 `method_splitter()` 和 `*args` 接受 `**kwargs` 函数参数并把它们传递到正确的视图。*any* 但是在我们这样做之前，我们要调用两次获得参数 `kwargs.pop()``GET``POST` ，如果它们合法的话。 (我们通过指定pop的缺省值为None,来避

免由于一个或者多个关键字缺失带来的KeyError)

包装视图函数

我们最终的视图技巧利用了一个高级python技术。 假设你发现自己在各个不同视图里重复了大量代码，就像 这个例子：

```
def my_view1(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render_to_response('template1.html')

def my_view2(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render_to_response('template2.html')

def my_view3(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/')
    # ...
    return render_to_response('template3.html')
```

这里，每一个视图开始都检查 `request.user` 是否是已经认证的，是的话，当前用户已经成功登陆站点否则就重定向 `/accounts/login/` (注意,虽然我们还没有讲到request.user,但是14章将要讲到它.就如你所想像的,request.user描述当前用户是登陆的还是匿名)

如果我们能够从每个视图里移除那些 重复代，并且只在需要认证的时候指明它们，那就完美了。 我们能够通过使用一个视图包装达到目的。 花点时间来看看这个：

```
def requires_login(view):
    def new_view(request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseRedirect('/accounts/login/')
        return view(request, *args, **kwargs)
    return new_view
```

函数requires_login,传入一个视图函数view,然后返回一个新的视图函数new_view.这个新的视图函数new_view在函数requires_login内定义 处理request.user.is_authenticated()这个验证,从而决定是否执行原来的view函数

现在,我们可以从views中去掉if not request.user.is_authenticated()验证.我们可以在URLconf中很容易的用requires_login来包装实现.

```
from django.conf.urls.defaults import *
from mysite.views import requires_login, my_view1, my_view2, my_view3

urlpatterns = patterns('',
    (r'^view1/$', requires_login(my_view1)),
    (r'^view2/$', requires_login(my_view2)),
    (r'^view3/$', requires_login(my_view3)),
)
```

优化后的代码和前面的功能一样,但是减少了代码冗余 现在我们建立了一个漂亮,通用的函数requires_login()来帮助我们修饰所有需要它来验证的视图

包含其他URLconf

如果你试图让你的代码用在多个基于Django的站点上，你应该考虑将你的URLconf以包含的方式来处理。

在任何时候，你的URLconf都可以包含其他URLconf模块。 对于根目录是基于一系列URL的站点来说，这是必要的。 例如下面的，URLconf包含了其他URLConf：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^weblog/', include('mysite.blog.urls')),
    (r'^photos/', include('mysite.photos.urls')),
    (r'^about/$', 'mysite.views.about'),
)
```

在前面第6章介绍Django的admin模块时我们曾经见过include. admin模块有他自己的URLconf,你仅仅只需要在你自己的代码中加入include就可以了.

这里有个很重要的地方： 例子中的指向 `include()` 的正则表达式并 不 包含一个 `$` （字符串结尾匹配符），但是包含了一个斜杆。 每当Django遇到 `include()` 时，它将截断匹配的URL，并把剩余的字符串发往包含的URLconf作进一步处理。

继续看这个例子，这里就是被包含的URLconf `mysite.blog.urls` ：

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^(\d\d\d\d)/$', 'mysite.blog.views.year_detail'),
    (r'^(\d\d\d\d)/(\d\d)/$', 'mysite.blog.views.month_detail'),
)
```

通过这两个URLconf，下面是一些处理请求的例子：

- `/weblog/2007/` ：在第一个URLconf中，模式 `r'^weblog/'` 被匹配。 因为它是一个 `include()` ，Django将截掉所有匹配的文本，在这里是 `'weblog/'` 。URL剩余的部分是 `2007/` ， 将在 `mysite.blog.urls` 这个URLconf的第一行中被匹配到。 URL仍存在的部

分为 `2007/` ,与第一行的 `mysite.blog.urls` URL设置相匹配。

- /weblog//2007/(包含两个斜杠) 在第一个URLconf中,r'^weblog/'匹配 因为它有一个include(),django去掉了匹配的部,在这个例子中匹配的部分是'weblog/' 剩下的部分是/2007/ (最前面有一个斜杠),不匹配mysite.blog.urls中的任何一行.
- `/about/` : 这个匹配第一个URLconf中的 `mysite.views.about` 视图。

捕获的参数如何和**include()**协同工作

一个被包含的URLconf接收任何来自parent URLconfs的被捕获的参数，比如:

```
# root urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
)

# foo/urls/blog.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'foo.views.blog_index'),
    (r'^archive/$', 'foo.views.blog_archive'),
)
```

在这个例子中，被捕获的 `username` 变量将传递给被包含的 URLconf，进而传递给那个URLconf中的 每一个 视图函数。

注意，这个被捕获的参数 总是 传递到被包含的URLconf中的 每一 行，不管那些行对应的视图是否需要这些参数。 因此，这个技术只有在你确实需要那个被传递的参数的时候才显得有用。

额外的**URLconf**如何和**include()**协同工作

相似的，你可以传递额外的URLconf选项到 `include()` , 就像你可以通过字典传递额外的URLconf选项到普通的视图。 当你这样做的时候，被包含URLconf的 每一 行都会收到那些额外的参数。

比如，下面的两个URLconf在功能上是相等的。

第一个：

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner'), {'blogid': 3}),
)

# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
    (r'^rss/$', 'mysite.views.rss'),
)
```

第二个

```
# urls.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^blog/', include('inner')),
)

# inner.py

from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive', {'blogid': 3}),
    (r'^about/$', 'mysite.views.about', {'blogid': 3}),
    (r'^rss/$', 'mysite.views.rss', {'blogid': 3}),
)
```

这个例子和前面关于被捕获的参数一样（在上一节就解释过这一点），额外的选项将 总是 被传递到被包含的URLconf中的 每一 行，不管那一行对应的视图是否确实作为有效参数接收这些选项，因此，这个技术只有在你确实需要那个被传递的额外参数的时候才显得有用。 因为这个原因，这种技术仅当你确信在涉及到的接受到额外你给出的选项的每个URLconf时有用的才奏效。

下一章

这一章提供了很多高级视图和URLconfs的小提示和技巧。 接下来，在Chapter 9,我们将会将这个先进的处理方案带给djangos模板系统。

第九章：模板高级进阶

虽然大多数和Django模板语言的交互都是模板作者的工作，但你可能想定制和扩展模板引擎，让它做一些它不能做的事情，或者是以其他方式让你的工作更轻松。

本章深入探讨Django的模板系统。 如果你想扩展模板系统或者只是对它的工作原理感觉到好奇，本章涉及了你需要了解的东西。 它也包含一个自动转意特征，如果你继续使用django，随着时间的推移你一定会注意这个安全考虑。

如果你想把Django的模版系统作为另外一个应用程序的一部分（就是说，仅使用Django的模板系统而不使用Django框架的其他部分），那你一定要读一下"配置独立模式下的模版系统"这一节。

模板语言回顾

首先，让我们快速回顾一下第四章介绍的若干专业术语：

> 模板 是一个纯文本文件，或是一个用Django模板语言标记过的普通的Python字符串。模板可以包含模板标签和变量。
>
> 模板标签 是在一个模板里面起作用的的标记。 这个定义故意搞得模糊不清。 例如，一个模版标签能够产生作为控制结构的内容（一个 `if` 语句或 `for` 循环), 可以获取数据库内容，或者访问其他的模板标签。
>
> 区块标签被 `{%` 和 `%}` 包围：

```
{% if is_logged_in %}
    Thanks for logging in!
{% else %}
    Please log in.
{% endif %}
```

> 变量 是一个在模板里用来输出值的标记。
>
> 变量标签被 `{{` 和 `}}` 包围：

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

> *context* 是一个传递给模板的名称到值的映射（类似Python字典）。
>
> 模板 渲染 就是是通过从context获取值来替换模板中变量并执行所有的模板标签。

关于这些基本概念更详细的内容，请参考第四章。

本章的其余部分讨论了扩展模板引擎的方法。 首先，我们快速的看一下第四章遗留的内容。

RequestContext和Context处理器

你需要一段context来解析模板。 一般情况下，这是一个 `django.template.Context` 的实例，不过在Django中还可以用一个特殊的子类， `django.template.RequestContext` ，这个用起来稍微有些不同。 `RequestContext` 默认地在模板context中加入了一些变量，如 `HttpRequest` 对象或当前登录用户的相关信息。

当你不想在一系例模板中都明确指定一些相同的变量时，你应该使用 `RequestContext` 。 例如，考虑这两个视图：

```
from django.template import loader, Context

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am view 1.'
    })
    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am the second view.'
    })
    return t.render(c)
```

（注意，在这些例子中，我们故意 不 使用 `render_to_response()` 这个快捷方法，而选择手动载入模板，手动构造context对象然后渲染模板。 是为了能够清晰的说明所有步骤。）

每个视图都给模板传入了三个相同的变量： `app` 、 `user` 和 `ip_address` 。 如果我们把这些冗余去掉会不会更好？

创建 `RequestContext` 和 **context**处理器 就是为了解决这个问题。 Context处理器允许你设置一些变量，它们会在每个context中自动被设置好，而不必每次调用 `render_to_response()` 时都指定。 要点就是，当你渲染模板时，你要用 `RequestContext` 而不是 `Context` 。

最直接的做法是用context处理器来创建一些处理器并传递给 `RequestContext` 。上面的例子可以用context processors改写如下：

```
from django.template import loader, RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = RequestContext(request, {'message': 'I am view 1.'},
            processors=[custom_proc])
    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = RequestContext(request, {'message': 'I am the second view.'},
            processors=[custom_proc])
    return t.render(c)
```

我们来通读一下代码：

- 首先，我们定义一个函数 `custom_proc` 。这是一个context处理器，它接收一个 `HttpRequest` 对象，然后返回一个字典，这个字典中包含了可以在模板context中使用的变量。 它就做了这么多。

- 我们在这两个视图函数中用 `RequestContext` 代替了 `Context` 。在context对象的构建上有两个不同点。 一， `RequestContext` 的第一个参数需要传递一个 `HttpRequest` 对象，就是传递给视图函数的第一个参数（ `request` ）。二， `RequestContext` 有一个可选的参数 `processors` ，这是一个包含context处理器函数的列表或者元组。 在这里，我们传递了我们之前定义的处理器函数 `curstom_proc` 。

- 每个视图的context结构里不再包含 `app` 、 `user` 、 `ip_address` 等变量，因为这些由 `custom_proc` 函数提供了。

- 每个视图 仍然 具有很大的灵活性，可以引入我们需要的任何模板变量。 在这个例子中， `message` 模板变量在每个视图中都不一样。

在第四章，我们介绍了 `render_to_response()` 这个快捷方式，它可以简化调用 `loader.get_template()` ,然后创建一个 `Context` 对象，最后再调用模板对象的 `render()` 过程。 为了讲解context处理器底层是如何工作的，在上面的例子中我们没有使用 `render_to_response()` 。但是建议选择 `render_to_response()` 作为context的处理器。这就需要用到 `context_instance` 参数：

```
from django.shortcuts import render_to_response
from django.template import RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    # ...
    return render_to_response('template1.html',
        {'message': 'I am view 1.'},
        context_instance=RequestContext(request, processors=[custom_proc]))

def view_2(request):
    # ...
    return render_to_response('template2.html',
        {'message': 'I am the second view.'},
        context_instance=RequestContext(request, processors=[custom_proc]))
```

在这，我们将每个视图的模板渲染代码写成了一个单行。

虽然这是一种改进，但是，请考虑一下这段代码的简洁性，我们现在不得不承认的是在 另外一方面有些过分了。 我们以代码冗余（在 `processors` 调用中）的代价消除了数据上的冗余（我们的模板变量）。 由于你不得不一直键入 `processors` ，所以使用context处理器并没有减少太多的输入量。

Django因此提供对 全局 context处理器的支持。 `TEMPLATE_CONTEXT_PROCESSORS` 指定了哪些 `context processors` 总是默认被使用。这样就省去了每次使用 `RequestContext` 都指定 `processors` 的麻烦。

默认情况下， `TEMPLATE_CONTEXT_PROCESSORS` 设置如下：

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.core.context_processors.auth',
    'django.core.context_processors.debug',
    'django.core.context_processors.i18n',
    'django.core.context_processors.media',
)
```

这个设置项是一个可调用函数的元组，其中的每个函数使用了和上文中我们的 `custom_proc` 相同的接口，它们以request对象作为参数，返回一个会被合并传给context的字典： 接收一个request对象作为参数，返回一个包含了将被合并到context中的项的字典。

每个处理器将会按照顺序应用。 也就是说如果你在第一个处理器里面向context添加了一个变量，而第二个处理器添加了同样名字的变量，那么第二个将会覆盖第一个。

Django提供了几个简单的context处理器，有些在默认情况下被启用的。

django.core.context_processors.auth

如果 `TEMPLATE_CONTEXT_PROCESSORS` 包含了这个处理器，那么每个 `RequestContext` 将包含这些变量：

- `user` ：一个 `django.contrib.auth.models.User` 实例，描述了当前登录用户（或者一个 `AnonymousUser` 实例，如果客户端没有登录）。
- `messages` ：一个当前登录用户的消息列表（字符串）。 在后台，对每一个请求，这个变量都调用 `request.user.get_and_delete_messages()` 方法。 这个方法收集用户的消息然后把它们从数据库中删除。
- `perms` ： `django.core.context_processors.PermWrapper` 的一个实例，包含了当前登录用户有哪些权限。

关于users、permissions和messages的更多内容请参考第14章。

django.core.context_processors.debug

这个处理器把调试信息发送到模板层。 如果 `TEMPLATE_CONTEXT_PROCESSORS` 包含这个处理器，每一个 `RequestContext` 将包含这些变量：

- `debug` ：你设置的 `DEBUG` 的值（ `True` 或 `False` ）。你可以在模板里面用这个变量测试是否处在debug模式下。
- `sql_queries` ：包含类似于 ``{‘sql’: …, ‘time’: `` 的字典的一个列表， 记录了这个请求期间的每个SQL查询以及查询所耗费的时间。 这个列表是按照请求顺序进行排列的。

  System Message: WARNING/2 (`<string>` , line 315); *backlink*

  Inline literal start-string without end-string.

由于调试信息比较敏感，所以这个context处理器只有当同时满足下面两个条件的时候才有效：

- `DEBUG` 参数设置为 `True` 。
- 请求的ip应该包含在 `INTERNAL_IPS` 的设置里面。

细心的读者可能会注意到 `debug` 模板变量的值永远不可能为 `False` ，因为如果 `DEBUG` 是 `False` ，那么 `debug` 模板变量一开始就不会被RequestContext所包含。

django.core.context_processors.i18n

如果这个处理器启用，每个 `RequestContext` 将包含下面的变量：

- `LANGUAGES` ： `LANGUAGES` 选项的值。

- `LANGUAGE_CODE` ：如果 `request.LANGUAGE_CODE` 存在，就等于它；否则，等同于 `LANGUAGE_CODE` 设置。

附录E提供了有关这两个设置的更多的信息。

django.core.context_processors.request

如果启用这个处理器，每个 `RequestContext` 将包含变量 `request` ， 也就是当前的 `HttpRequest` 对象。 注意这个处理器默认是不启用的，你需要激活它。

如果你发现你的模板需要访问当前的 `HttpRequest` 你就需要使用它:

```
{{ request.REMOTE_ADDR }}
```

写Context处理器的一些建议

编写处理器的一些建议：

- 使每个context处理器完成尽可能小的功能。 使用多个处理器是很容易的，所以你可以根据逻辑块来分解功能以便将来复用。
- 要注意 `TEMPLATE_CONTEXT_PROCESSORS` 里的context processor 将会在基于这个settings.py的每个 模板中有效，所以变量的命名不要和模板的变量冲突。 变量名是大小写敏感的，所以processor的变量全用大写是个不错的主意。
- 不论它们存放在哪个物理路径下，只要在你的Python搜索路径中，你就可以在 `TEMPLATE_CONTEXT_PROCESSORS` 设置里指向它们。 建议你把它们放在应用或者工程目录下名为 `context_processors.py` 的文件里。

html自动转意

从模板生成html的时候，总是有一个风险——变量包了含会影响结果html的字符。 例如，考虑这个模板片段：

```
Hello, {{ name }}.
```

一开始，这看起来是显示用户名的一个无害的途径，但是考虑如果用户输入如下的名字将会发生什么：

```
<script>alert('hello')</script>
```

用这个用户名，模板将被渲染成：

```
Hello, <script>alert('hello')</script>
```

这意味着浏览器将弹出JavaScript警告框！

类似的，如果用户名包含 `小于` 符号，就像这样：

用户名

那样的话模板结果被翻译成这样：

```
Hello, <b>username
```

页面的剩余部分变成了粗体！

显然，用户提交的数据不应该被盲目信任，直接插入到你的页面中。因为一个潜在的恶意的用户能够利用这类漏洞做坏事。 这类漏洞称为被跨域脚本 (XSS) 攻击。 关于安全的更多内容，请看20章

为了避免这个问题，你有两个选择：

- 一是你可以确保每一个不被信任的变量都被 `escape` 过滤器处理一遍，把潜在有害的html字符转换为无害的。 这是最初几年Django的默认方案，但是这样做的问题是它把责任推给你（开发者、模版作者）自己，来确保把所有东西转意。 很容易就忘记转意数据。
- 二是，你可以利用Django的自动html转意。 这一章的剩余部分描述自动转意是如何工作的。

在django里默认情况下，每一个模板自动转意每一个变量标签的输出。 尤其是这五个字符。

- ```\ ```

  System Message: WARNING/2 (`<string>` , line 491); *backlink*

  Inline literal start-string without end-string.

- > 被转换为>
- `'` （单引号）被转换为 `'`
- `"` (双引号)被转换为 `"`
- `&` is converted to `&`

另外，我强调一下这个行为默认是开启的。 如果你正在使用django的模板系统，那么你是被保护的。

如何关闭它

如果你不想数据被自动转意，在每一站点级别、每一模板级别或者每一变量级别你都有几种方法来关闭它。

为什么要关闭它？ 因为有时候模板变量包含了一些原始html数据，在这种情况下我们不想它们的内容被转意。 例如，你可能在数据库里存储了一段被信任的html代码，并且你想直接把它嵌入到你的模板里。 或者，你可能正在使用Django的模板系统生成非html文本，比如一封e-mail。

对于单独的变量

用safe过滤器为单独的变量关闭自动转意：

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

你可以把*safe*当做*safe from further escaping*的简写，或者当做可以被直接译成*HTML*的内容。在这个例子里，如果 `数据` 包含 `''`，那么输出会变成：

```
This will be escaped: &lt;b&gt;
This will not be escaped: <b>
```

对于模板块

为了控制模板的自动转意,用标签autoescape来包装整个模板(或者模板中常用的部分),就像这样：

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

autoescape 标签有两个参数on和off 有时,你可能想阻止一部分自动转意,对另一部分自动转意。 这是一个模板的例子：

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be auto-escaped: {{ data }}.

    Nor this: {{ other_data }}
    {% autoescape on %}
        Auto-escaping applies again: {{ name }}
    {% endautoescape %}
{% endautoescape %}
```

auto-escaping 标签的作用域不仅可以影响到当前模板还可以通过include标签作用到其他标签,就像block标签一样。 例如：

```
# base.html

{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}

# child.html

{% extends "base.html" %}
{% block title %}This & that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

由于在base模板中自动转意被关闭,所以在child模板中自动转意也会关闭.因此,在下面一段HTML被提交时,变量greeting的值就为字符串Hello!

```
<h1>This & that</h1>
<b>Hello!</b>
```

备注

通常,模板作者没必要为自动转意担心. 基于Pyhton的开发者(编写VIEWS视图和自定义过滤器)只需要考虑哪些数据不需要被转意,适时的标记数据,就可以让它们在模板中工作。

如果你正在编写一个模板而不知道是否要关闭自动转意,那就为所有需要转意的变量添加一个escape过滤器。 当自动转意开启时，使用escape过滤器似乎会两次转意数据，但其实没有任何危险。因为escape过滤器不作用于被转意过的变量。

过滤器参数里的字符串常量的自动转义

就像我们前面提到的,过滤器也可以是字符串.

```
{{ data|default:"This is a string literal." }}
```

所有字符常量没有经过转义就被插入模板,就如同它们都经过了safe过滤。 这是由于字符常量完全由模板作者决定,因此编写模板的时候他们会确保文本的正确性。

这意味着你必须这样写

```
{{ data|default:"3 &lt; 2" }}
```

而不是这样

```
{{ data|default:"3 < 2" }}  <-- Bad! Don't do this.
```

这点对来自变量本身的数据不起作用。 如果必要,变量内容会自动转义,因为它们不在模板作者的控制下。

模板加载的内幕

一般说来，你会把模板以文件的方式存储在文件系统中，但是你也可以使用自定义的 *template loaders* 从其他来源加载模板。

Django有两种方法加载模板

- `django.template.loader.get_template(template_name)` ： `get_template` 根据给定的模板名称返回一个已编译的模板（一个 `Template` 对象）。 如果模板不存在，就触发 `TemplateDoesNotExist` 的异常。

- `django.template.loader.select_template(template_name_list)` ： `select_template` 很像 `get_template` ，不过它是以模板名称的列表作为参数的。 它会返回列表中存在的第一个模板。 如果模板都不存在，将会触发 `TemplateDoesNotExist` 异常。

正如在第四章中所提到的，默认情况下这些函数使用 `TEMPLATE_DIRS` 的设置来载入模板。 但是，在内部这些函数可以指定一个模板加载器来完成这些繁重的任务。

一些加载器默认被禁用，但是你可以通过编辑 `TEMPLATE_LOADERS` 设置来激活它们。 `TEMPLATE_LOADERS` 应当是一个字符串的元组，其中每个字符串都表示一个模板加载器。 这些模板加载器随Django一起发布。

> `django.template.loaders.filesystem.load_template_source` ：这个加载器根据 `TEMPLATE_DIRS` 的设置从文件系统加载模板。它默认是可用的。
>
> `django.template.loaders.app_directories.load_template_source` ：这个加 载器从文件系统上的Django应用中加载模板。 对 `INSTALLED_APPS` 中的每个应用，这个加载器会查找 `templates` 子目录。 如果这个目录存在，Django就在那里寻找模板。
>
> 这意味着你可以把模板和你的应用一起保存，从而使得Django应用更容易和默认模板一起发布。 例如，如果 `INSTALLED_APPS` 包含 `('myproject.polls','myproject.music')` ，那么 `get_template('foo.html')` 会按这个顺序查找模板：
>
> - `/path/to/myproject/polls/templates/foo.html`
> - `/path/to/myproject/music/templates/foo.html`
>
> 请注意加载器在首次被导入的时候会执行一个优化： 它会缓存一个列表，这个列表包含了 `INSTALLED_APPS` 中带有 `templates` 子目录的包。
>
> 这个加载器默认启用。
>
> `django.template.loaders.eggs.load_template_source` ：这个加载器类似 `app_directories` ，只不过它从Python eggs而不是文件系统中加载模板。 这个加载器默认被禁用；如果你使用eggs来发布你的应用，那么你就需要启用它。 Python eggs可以将Python代码压缩到一个文件中。

Django按照 `TEMPLATE_LOADERS` 设置中的顺序使用模板加载器。 它逐个使用每个加载器直至找到一个匹配的模板。

扩展模板系统

既然你已经对模板系统的内幕多了一些了解，让我们来看看如何使用自定义的代码来扩展这个系统吧。

绝大部分的模板定制是以自定义标签/过滤器的方式来完成的。 尽管Django模板语言自带了许多内建标签和过滤器，但是你可能还是需要组建你自己的标签和过滤器库来满足你的需要。 幸运的是，定义你自己的功能非常容易。

创建一个模板库

不管是写自定义标签还是过滤器，第一件要做的事是创建模板库（Django能够导入的基本结构）。

创建一个模板库分两步走：

> 第一，决定模板库应该放在哪个Django应用下。 如果你通过 `manage.py startapp` 创建了一个应用，你可以把它放在那里，或者你可以为模板库单独创建一个应用。 我们更推荐使用后者，因为你的filter可能在后来的工程中有用。
>
> 无论你采用何种方式，请确保把你的应用添加到 `INSTALLED_APPS` 中。 我们稍后会解释这一点。
>
> 第二，在适当的Django应用包里创建一个 `templatetags` 目录。 这个目录应当和 `models.py` 、 `views.py` 等处于同一层次。 例如：

```
books/
    __init__.py
    models.py
    templatetags/
    views.py
```

> 在 `templatetags` 中创建两个空文件： 一个 `__init__.py` （告诉Python这是 一个包含了Python代码的包）和一个用来存放你自定义的标签/过滤器定义的文件。 第二个文件的名字稍后将用来加载标签。 例如，如果你的自定义标签/过滤器在一个叫作 `poll_extras.py` 的文件中，你需要在模板中写入如下内容：

```
{% load poll_extras %}
```

> `{% load %}` 标签检查 `INSTALLED_APPS` 中的设置，仅允许加载已安装的Django应用程序中的模板库。 这是一个安全特性；它可以让你在一台电脑上部署很多的模板库的代码，而又不用把它们暴露给每一个Django安装。

如果你写了一个不和任何特定模型/视图关联的模板库，那么得到一个仅包含 `templatetags` 包的Django应用程序包是完全正常的。 对于在 `templatetags` 包中放置多少个模块没有做任何的限制。 需要了解的是： `{%load%}` 语句是通过指定的Python模块名而不是应用名来加载标签/过滤器的。

一旦创建了Python模块，你只需根据是要编写过滤器还是标签来相应的编写一些Python代码。

作为合法的标签库，模块需要包含一个名为 `register` 的模块级变量。这个变量是 `template.Library` 的实例，是所有注册标签和过滤器的数据结构。 所以，请在你的模块的顶部插入如下语句：

```
from django import template

register = template.Library()
```

注意

请阅读Django默认的过滤器和标签的源码，那里有大量的例子。 他们分别为：`django/template/defaultfilters.py` 和 django/template/defaulttags.py 。django.contrib中的某些应用程序也包含模板库。

创建 `register` 变量后，你就可以使用它来创建模板的过滤器和标签了。

自定义模板过滤器

自定义过滤器就是有一个或两个参数的Python函数:

- (输入)变量的值
- 参数的值， 可以是默认值或者完全留空

例如，在过滤器 `{{ var|foo:"bar" }}` 中 ，过滤器 `foo` 会被传入变量 `var` 和默认参数 `bar` 。

过滤器函数应该总有返回值。 而且不能触发异常，它们都应该静静地失败。 如果出现错误，应该返回一个原始输入或者空字符串，这会更有意义。

这里是一些定义过滤器的例子：

```
def cut(value, arg):
    "Removes all values of arg from the given string"
    return value.replace(arg, '')
```

下面是一个可以用来去掉变量值空格的过滤器例子：

```
{{ somevariable|cut:" " }}
```

大多数过滤器并不需要参数。 下面的例子把参数从你的函数中拿掉了：

```
def lower(value): # Only one argument.
    "Converts a string into all lowercase"
    return value.lower()
```

当你定义完过滤器后，你需要用 `Library` 实例来注册它，这样就能通过Django的模板语言来使用了：

```
register.filter('cut', cut)
register.filter('lower', lower)
```

`Library.filter()` 方法需要两个参数：

- 过滤器的名称（一个字串）
- 过滤器函数本身

如果你使用的是Python 2.4或者更新的版本，你可以使用装饰器 `register.filter()` ：

```
@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
def lower(value):
    return value.lower()
```

如果你想第二个例子那样不使用 `name` 参数，那么Django会把函数名当作过滤器的名字。

下面是一个完整的模板库的例子，它包含一个 `cut` 过滤器：

```
from django import template

register = template.Library()

@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')
```

自定义模板标签

标签要比过滤器复杂些，因为标签几乎能做任何事情。

第四章描述了模板系统的两步处理过程： 编译和呈现。 为了自定义一个模板标签，你需要告诉Django当遇到你的标签时怎样进行这个过程。

当Django编译一个模板时，它将原始模板分成一个个 节点 。每个节点都是 `django.template.Node` 的一个实例，并且具备 `render()` 方法。 于是，一个已编译的模板就是 `节点` 对象的一个列表。 例如，看看这个模板：

```
Hello, {{ person.name }}.

{% ifequal name.birthday today %}
    Happy birthday!
{% else %}
    Be sure to come back on your birthday
    for a splendid surprise message.
{% endifequal %}
```

被编译的模板表现为节点列表的形式：

- 文本节点： `"Hello, "`
- 变量节点： `person.name`
- 文本节点: `".\n\n"`
- IfEqual节点: `name.birthday` 和 `today`

当你调用一个已编译模板的 `render()` 方法时，模板就会用给定的context来调用每个在它的节点列表上的所有节点的 `render()` 方法。 这些渲染的结果合并起来，形成了模板的输出。因此，要自定义模板标签，你需要指明原始模板标签如何转换成 `节点` （编译函数）和节点的 `render()` 方法完成的功能 。

在下面的章节中，我们将详细解说写一个自定义标签时的所有步骤。

编写编译函数

当遇到一个模板标签（template tag）时，模板解析器就会把标签包含的内容，以及模板解析器自己作为参数调用一个python函数。 这个函数负责返回一个和当前模板标签内容相对应的节点（Node）的实例。

例如，写一个显示当前日期的模板标签：{% current_time %}。该标签会根据参数指定的strftime 格式（参见： `http://www.djangoproject.com/r/python/strftime/` ）显示当前时间。首先确定标签的语法是个好主意。 在这个例子里，标签应该这样使用：

```
<p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

注意

没错，这个模板标签是多余的，Django默认的 `{% now %}` 用更简单的语法完成了同样的工作。 这个模板标签在这里只是作为一个例子。

这个函数的分析器会获取参数并创建一个 `Node` 对象:

```
from django import template

register = template.Library()

def do_current_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, format_string = token.split_contents()
    except ValueError:
        msg = '%r tag requires a single argument' % token.split_contents()[0]
        raise template.TemplateSyntaxError(msg)
    return CurrentTimeNode(format_string[1:-1])
```

这里需要说明的地方很多：

- 每个标签编译函数有两个参数， `parser` 和 `token` 。 `parser` 是模板解析器对象。 我们在这个例子中并不使用它。 `token` 是正在被解析的语句。
- `token.contents` 是包含有标签原始内容的字符串。 在我们的例子中，它是 `'current_time "%Y-%m-%d %I:%M %p"'` 。

- `token.split_contents()` 方法按空格拆分参数同时保证引号中的字符串不拆分。 应该避免使用 `token.contents.split()` （仅使用Python的标准字符串拆分）。 它不够健壮，因为它只是简单的按照所有空格进行拆分，包括那些引号引起来的字符串中的空格。
- 这个函数可以抛出 `django.template.TemplateSyntaxError` ，这个异常提供所有语法错误的有用信息。
- 不要把标签名称硬编码在你的错误信息中，因为这样会把标签名称和你的函数耦合在一起。 `token.split_contents()[0]` 总是记录标签的名字，就算标签没有任何参数。
- 这个函数返回一个 `CurrentTimeNode` （稍后我们将创建它），它包含了节点需要知道的关于这个标签的全部信息。 在这个例子中，它只是传递了参数 `"%Y-%m-%d %I:%M %p"` 。模板标签开头和结尾的引号使用 `format_string[1:-1]` 除去。
- 模板标签编译函数 必须 返回一个 `Node` 子类，返回其它值都是错的。

编写模板节点

编写自定义标签的第二步就是定义一个拥有 `render()` 方法的 `Node` 子类。 继续前面的例子，我们需要定义 `CurrentTimeNode` ：

```
import datetime

class CurrentTimeNode(template.Node):
    def __init__(self, format_string):
        self.format_string = str(format_string)

    def render(self, context):
        now = datetime.datetime.now()
        return now.strftime(self.format_string)
```

这两个函数（ `__init__()` 和 `render()` ）与模板处理中的两步（编译与渲染）直接对应。这样，初始化函数仅仅需要存储后面要用到的格式字符串，而 `render()` 函数才做真正的工作。

与模板过滤器一样，这些渲染函数应该静静地捕获错误，而不是抛出错误。 模板标签只允许在编译的时候抛出错误。

注册标签

最后，你需要用你模块的 `Library` 实例注册这个标签。 注册自定义标签与注册自定义过滤器非常类似（如前文所述）。 只需实例化一个 `template.Library` 实例然后调用它的 `tag()` 方法。 例如：

```
register.tag('current_time', do_current_time)
```

`tag()` 方法需要两个参数:

- 模板标签的名字（字符串）。
- 编译函数。

和注册过滤器类似，也可以在Python2.4及其以上版本中使用 `register.tag` 装饰器：

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    # ...

@register.tag
def shout(parser, token):
    # ...
```

如果你像在第二个例子中那样忽略 `name` 参数的话，Django会使用函数名称作为标签名称。

在上下文中设置变量

前一节的例子只是简单的返回一个值。 很多时候设置一个模板变量而非返回值也很有用。 那样，模板作者就只能使用你的模板标签所设置的变量。

要在上下文中设置变量，在 `render()` 函数的context对象上使用字典赋值。 这里是一个修改过的 `CurrentTimeNode` ，其中设定了一个模板变量 `current_time` ，并没有返回它：

```
class CurrentTimeNode2(template.Node):
    def __init__(self, format_string):
        self.format_string = str(format_string)

    def render(self, context):
        now = datetime.datetime.now()
        context['current_time'] = now.strftime(self.format_string)
        return ''
```

(我们把创建函数 `do_current_time2` 和注册给 `current_time2` 模板标签的工作留作读者练习。)

注意 `render()` 返回了一个空字符串。 `render()` 应当总是返回一个字符串，所以如果模板标签只是要设置变量， `render()` 就应该返回一个空字符串。

你应该这样使用这个新版本的标签：

```
{% current_time2 "%Y-%M-%d %I:%M %p" %}
<p>The time is {{ current_time }}.</p>
```

但是 `CurrentTimeNode2` 有一个问题: 变量名 `current_time` 是硬编码的。 这意味着你必须确定你的模板在其它任何地方都不使用 `{{ current_time }}` ，因为 `{% current_time2 %}` 会盲目的覆盖该变量的值。

一种更简洁的方案是由模板标签来指定需要设定的变量的名称，就像这样：

```
{% get_current_time "%Y-%M-%d %I:%M %p" as my_current_time %}
<p>The current time is {{ my_current_time }}.</p>
```

为此，你需要重构编译函数和 `Node` 类，如下所示：

```
import re

class CurrentTimeNode3(template.Node):
    def __init__(self, format_string, var_name):
        self.format_string = str(format_string)
        self.var_name = var_name

    def render(self, context):
        now = datetime.datetime.now()
        context[self.var_name] = now.strftime(self.format_string)
        return ''

def do_current_time(parser, token):
    # This version uses a regular expression to parse tag contents.
    try:
        # Splitting by None == splitting by spaces.
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        msg = '%r tag requires arguments' % token.contents[0]
        raise template.TemplateSyntaxError(msg)

    m = re.search(r'(.*?) as (\w+)', arg)
    if m:
        fmt, var_name = m.groups()
    else:
        msg = '%r tag had invalid arguments' % tag_name
        raise template.TemplateSyntaxError(msg)

    if not (fmt[0] == fmt[-1] and fmt[0] in ('"', "'")):
        msg = "%r tag's argument should be in quotes" % tag_name
        raise template.TemplateSyntaxError(msg)

    return CurrentTimeNode3(fmt[1:-1], var_name)
```

现在 `do_current_time()` 把格式字符串和变量名传递给 `CurrentTimeNode3` 。

分析直至另一个模板标签

模板标签可以像包含其它标签的块一样工作（想想 `{% if %}` 、 `{% for %}` 等）。 要创建一个这样的模板标签，在你的编译函数中使用 `parser.parse()` 。

标准的 `{% comment %}` 标签是这样实现的：

```
def do_comment(parser, token):
    nodelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''
```

`parser.parse()` 接收一个包含了需要分析的模板标签名的元组作为参数。 它返回一个 `django.template.NodeList` 实例，它是一个包含了所有*Node*对象的列表，这些对象是解析器在解析到任一元组中指定的标签之前遇到的内容.

因此在前面的例子中， `nodelist` 是在 `{% comment %}` 和 `{% endcomment %}` 之间所有节点的列表，不包括 `{% comment %}` 和 `{% endcomment %}` 自身。

在 `parser.parse()` 被调用之后，分析器还没有清除 `{% endcomment %}` 标签，因此代码需要显式地调用 `parser.delete_first_token()` 来防止该标签被处理两次。

之后 `CommentNode.render()` 只是简单地返回一个空字符串。 在 `{% comment %}` 和 `{% endcomment %}` 之间的所有内容都被忽略。

分析直至另外一个模板标签并保存内容

在前一个例子中， `do_comment()` 抛弃了 `{% comment %}` 和 `{% endcomment %}` 之间的所有内容。当然也可以修改和利用下标签之间的这些内容。

例如，这个自定义模板标签 `{% upper %}` ，它会把它自己和 `{% endupper %}` 之间的内容变成大写：

```
{% upper %}
    This will appear in uppercase, {{ user_name }}.
{% endupper %}
```

就像前面的例子一样，我们将使用 `parser.parse()` 。这次，我们将产生的 `nodelist` 传递给 `Node` ：

```
def do_upper(parser, token):
    nodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(nodelist)

class UpperNode(template.Node):
    def __init__(self, nodelist):
        self.nodelist = nodelist

    def render(self, context):
        output = self.nodelist.render(context)
        return output.upper()
```

这里唯一的一个新概念是 `UpperNode.render()` 中的 `self.nodelist.render(context)` 。它对节点列表中的每个 `Node` 简单的调用 `render()` 。

更多的复杂渲染示例请查看 `django/template/defaulttags.py` 中的 `{% if %}` 、 `{% for %}` 、 `{% ifequal %}` 和 `{% ifchanged %}` 的代码。

简单标签的快捷方式

许多模板标签接收单一的字符串参数或者一个模板变量引用，然后独立地根据输入变量和一些其它外部信息进行处理并返回一个字符串。 例如，我们先前写的 `current_time` 标签就是这样一个例子。 我们给定了一个格式化字符串，然后它返回一个字符串形式的时间。

为了简化这类标签，Django提供了一个帮助函数 `simple_tag` 。这个函数是 `django.template.Library` 的一个方法，它接受一个只有一个参数的函数作参数，把它包装在render函数和之前提及过的其他的必要单位中，然后通过模板系统注册标签。

我们之前的的 `current_time` 函数于是可以写成这样：

```
def current_time(format_string):
    try:
        return datetime.datetime.now().strftime(str(format_string))
    except UnicodeEncodeError:
        return ''

register.simple_tag(current_time)
```

在Python 2.4中，也可以使用装饰器语法：

```
@register.simple_tag
def current_time(token):
    # ...
```

有关 `simple_tag` 辅助函数，需要注意下面一些事情：

- 传递给我们的函数的只有（单个）参数。
- 在我们的函数被调用的时候，检查必需参数个数的工作已经完成了，所以我们不需要再做这个工作。
- 参数两边的引号（如果有的话）已经被截掉了，所以我们会接收到一个普通Unicode字符串。

包含标签

另外一类常用的模板标签是通过渲染 其他 模板显示数据的。 比如说，Django的后台管理界面，它使用了自定义的模板标签来显示新增/编辑表单页面下部的按钮。 那些按钮看起来总是一样的，但是链接却随着所编辑的对象的不同而改变。 这就是一个使用小模板很好的例子，这些小模板就是当前对象的详细信息。

这些排序标签被称为 包含标签 。如何写包含标签最好通过举例来说明。 让我们来写一个能够产生指定 `作者` 对象的书籍清单的标签。 我们将这样利用标签：

```
{% books_for_author author %}
```

结果将会像下面这样：

```
<ul>
    <li>The Cat In The Hat</li>
    <li>Hop On Pop</li>
    <li>Green Eggs And Ham</li>
</ul>
```

首先，我们定义一个函数，通过给定的参数生成一个字典形式的结果。 需要注意的是，我们只需要返回字典类型的结果就行了，不需要返回更复杂的东西。 这将被用来作为模板片段的内容：

```
def books_for_author(author):
    books = Book.objects.filter(authors__id=author.id)
    return {'books': books}
```

接下来，我们创建用于渲染标签输出的模板。 在我们的例子中，模板很简单：

```
<ul>
{% for book in books %}
    <li>{{ book.title }}</li>
{% endfor %}
</ul>
```

最后，我们通过对一个 `Library` 对象使用 `inclusion_tag()` 方法来创建并注册这个包含标签。

在我们的例子中，如果先前的模板在 `polls/result_snippet.html` 文件中，那么我们这样注册标签：

```
register.inclusion_tag('book_snippet.html')(books_for_author)
```

Python 2.4装饰器语法也能正常工作，所以我们可以这样写：

```
@register.inclusion_tag('book_snippet.html')
def books_for_author(author):
    # ...
```

有时候，你的包含标签需要访问父模板的context。 为了解决这个问题，Django为包含标签提供了一个 `takes_context` 选项。 如果你在创建模板标签时，指明了这个选项，这个标签就不需要参数，并且下面的Python函数会带一个参数： 就是当这个标签被调用时的模板context。

例如，你正在写一个包含标签，该标签包含有指向主页的 `home_link` 和 `home_title` 变量。 Python函数会像这样：

```
@register.inclusion_tag('link.html', takes_context=True)
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
```

（注意函数的第一个参数 必须 是 `context` 。）

模板 `link.html` 可能包含下面的东西：

```
Jump directly to <a href="{{ link }}">{{ title }}</a>.
```

然后您想使用自定义标签时，就可以加载它的库，然后不带参数地调用它，就像这样：

```
{% jump_link %}
```

编写自定义模板加载器

Djangos 内置的模板加载器（在先前的模板加载内幕章节有叙述）通常会满足你的所有的模板加载需求，但是如果你有特殊的加载需求的话，编写自己的模板加载器也会相当简单。 比如：你可以从数据库中，或者利用Python的绑定直接从Subversion库中，更或者从一个ZIP文档中加载模板。

模板加载器，也就是 `TEMPLATE_LOADERS` 中的每一项，都要能被下面这个接口调用：

```
load_template_source(template_name, template_dirs=None)
```

参数 `template_name` 是所加载模板的名称 (和传递给 `loader.get_template()` 或者 `loader.select_template()` 一样), 而 `template_dirs` 是一个可选的代替 `TEMPLATE_DIRS` 的搜索目录列表。

如果加载器能够成功加载一个模板, 它应当返回一个元组： `(template_source, template_path)` 。在这里的 `template_source` 就是将被模板引擎编译的的模板字符串，而 `template_path` 是被加载的模板的路径。 由于那个路径可能会出于调试目的显示给用户，因此它应当很快的指明模板从哪里加载。

如果加载器加载模板失败，那么就会触发 `django.template.TemplateDoesNotExist` 异常。

每个加载函数都应该有一个名为 `is_usable` 的函数属性。 这个属性是一个布尔值，用于告知模板引擎这个加载器是否在当前安装的Python中可用。 例如，如果 `pkg_resources` 模块没有安装的话，eggs加载器（它能够从python eggs中加载模板）就应该把 `is_usable` 设为 `False` ，因为必须通过 `pkg_resources` 才能从eggs中读取数据。

一个例子可以清晰地阐明一切。 这儿是一个模板加载函数，它可以从ZIP文件中加载模板。 它使用了自定义的设置 `TEMPLATE_ZIP_FILES` 来取代了 `TEMPLATE_DIRS` 用作查找路径，并且它假设在此路径上的每一个文件都是包含模板的ZIP文件：

```
from django.conf import settings
from django.template import TemplateDoesNotExist
import zipfile

def load_template_source(template_name, template_dirs=None):
    "Template loader that loads templates from a ZIP file."

    template_zipfiles = getattr(settings, "TEMPLATE_ZIP_FILES", [])

    # Try each ZIP file in TEMPLATE_ZIP_FILES.
    for fname in template_zipfiles:
        try:
            z = zipfile.ZipFile(fname)
            source = z.read(template_name)
        except (IOError, KeyError):
            continue
        z.close()
        # We found a template, so return the source.
        template_path = "%s:%s" % (fname, template_name)
        return (source, template_path)

    # If we reach here, the template couldn't be loaded
    raise TemplateDoesNotExist(template_name)

# This loader is always usable (since zipfile is included with Python)
load_template_source.is_usable = True
```

我们要想使用它，还差最后一步，就是把它加入到 `TEMPLATE_LOADERS` 。 如果我们将这个代码放入一个叫mysite.zip_loader的包中，那么我们要把mysite.zip_loader.load_template_source加到TEMPLATE_LOADERS中。

配置独立模式下的模板系统

注意：

这部分只针对于对在其他应用中使用模版系统作为输出组件感兴趣的人。 如果你是在Django应用中使用模版系统，请略过此部分。

通常，Django会从它的默认配置文件和由 `DJANGO_SETTINGS_MODULE` 环境变量所指定的模块中加载它需要的所有配置信息。 （这点在第四章的”特殊的Python命令提示行”一节解释过。） 但是当你想在非Django应用中使用模版系统的时候，采用环境变量并不方便，因为你可能更想同其余的应用一起配置你的模板系统，而不是处理配置文件并通过环境变量指向他们。

为了解决这个问题，你需要使用附录D中所描述的手动配置选项。概括的说，你需要导入正确的模板中的片段，然后在你访问任一个模板函数之前，首先用你想指定的配置访问Django.conf.settings.configure()。

你可能会考虑至少要设置 `TEMPLATE_DIRS` （如果你打算使用模板加载器）， `DEFAULT_CHARSET` （尽管默认的 `utf-8` 编码相当好用），以及 `TEMPLATE_DEBUG` 。所有可用的选项在附录D中都有详细描述，所有以 `TEMPLATE_` 开头的选项都可能使你感兴趣。

接下来做什么？

延续本章的高级话题，下一章 会继续讨论Django模版的高级用法。

第十章：数据模型高级进阶

在第5章里，我们介绍了Django的数据层如何定义数据模型以及如何使用数据库API来创建、检索、更新以及删除记录 在这章里，我们将向你介绍Django在这方面的一些更高级功能。

相关对象

先让我们回忆一下在第五章里的关于书本(book)的数据模型：

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __unicode__(self):
        return self.name

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

    def __unicode__(self):
        return u'%s %s' % (self.first_name, self.last_name)

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    def __unicode__(self):
        return self.title
```

如我们在第5章的讲解,获取数据库对象的特定字段的值只需直接使用属性。 例如,要确定ID为50的书本的标题,我们这样做:

```
>>> from mysite.books.models import Book
>>> b = Book.objects.get(id=50)
>>> b.title
u'The Django Book'
```

但是,在之前有一件我们没提及到的是表现为 `ForeignKey` 或 `ManyToManyField` 的关联对象字段,它们的作用稍有不同。

访问外键(Foreign Key)值

当你获取一个 `ForeignKey` 字段时,你会得到相关的数据模型对象。 例如:

```
>>> b = Book.objects.get(id=50)
>>> b.publisher
<Publisher: Apress Publishing>
>>> b.publisher.website
u'http://www.apress.com/'
```

对于用 `ForeignKey` 来定义的关系来说，在关系的另一端也能反向的追溯回来，只不过由于不对称性的关系而稍有不同。 通过一个 `publisher` 对象，直接获取 books ，用 publisher.book_set.all() ，如下：

```
>>> p = Publisher.objects.get(name='Apress Publishing')
>>> p.book_set.all()
[<Book: The Django Book>, <Book: Dive Into Python>, ...]
```

实际上， `book_set` 只是一个 `QuerySet` （参考第5章的介绍），所以它可以像 `QuerySet` 一样，能实现数据过滤和分切，例如：

```
>>> p = Publisher.objects.get(name='Apress Publishing')
>>> p.book_set.filter(name__icontains='django')
[<Book: The Django Book>, <Book: Pro Django>]
```

属性名称 `book_set` 是由模型名称的小写(如book)加 `_set` 组成的。

访问多对多值(Many-to-Many Values)

多对多和外键工作方式相同，只不过我们处理的是 `QuerySet` 而不是模型实例。 例如,这里是如何查看书籍的作者：

```
>>> b = Book.objects.get(id=50)
>>> b.authors.all()
[<Author: Adrian Holovaty>, <Author: Jacob Kaplan-Moss>]
>>> b.authors.filter(first_name='Adrian')
[<Author: Adrian Holovaty>]
>>> b.authors.filter(first_name='Adam')
[]
```

反向查询也可以。 要查看一个作者的所有书籍,使用 `author.book_set` ,就如这样:

```
>>> a = Author.objects.get(first_name='Adrian', last_name='Holovaty')
>>> a.book_set.all()
[<Book: The Django Book>, <Book: Adrian's Other Book>]
```

这里,就像使用 `ForeignKey` 字段一样，属性名 `book_set` 是在数据模型(model)名后追加 `_set` 。

更改数据库模式(Database Schema)

在我们在第5章介绍 `syncdb` 这个命令时, 我们注意到 `syncdb` 仅仅创建数据库里还没有的表， 它 并不 对你数据模型的修改进行同步,也不处理数据模型的删除。 如果你新增或修改数据模型里的字段,或是删除了一个数据模型，你需要手动在数据库里进行相应的修改。 这段将解释了具体怎么做：

当处理模型修改的时候，将Django的数据库层的工作流程铭记于心是很重要的。

- 如果模型包含一个未曾在数据库里建立的字段，Django会报出错信息。 当你第一次用Django的数据库API请求表中不存在的字段时会导致错误（就是说，它会在运行时出错，而不是编译时）。
- Django不关心数据库表中是否存在未在模型中定义的列。
- Django不关心数据库中是否存在未被模型表示的表格。

改变模型的模式架构意味着需要按照顺序更改Python代码和数据库。

添加字段

当要向一个产品设置表(或者说是model)添加一个字段的时候，要使用的技巧是利用Django不关心表里是否包含model里所没有的列的特性。 策略就是现在数据库里加入字段，然后同步Django的模型以包含新字段。

然而 这里有一个鸡生蛋蛋生鸡的问题 ,由于要想了解新增列的SQL语句，你需要使用Django的 `manage.py sqlall` 命令进行查看 ,而这又需要字段已经在模型里存在了。 (注意:你并 不是非得使用与Django相同的SQL语句创建新的字段，但是这样做确实是一个好主意 ,它能让一切都保持同步。)

这个鸡-蛋的问题的解决方法是在开发者环境里而不是发布环境里实现这个变化。 (你正使用的是测试/开发环境，对吧?)下面是具体的实施步骤。

首先，进入开发环境(也就是说，不是在发布环境里)：

1. 在你的模型里添加字段。
2. 运行 `manage.py sqlall [yourapp]` 来测试模型新的 `CREATE TABLE` 语句。 注意为新字段的列定义。
3. 开启你的数据库的交互命令界面(比如, `psql` 或 `mysql` , 或者可以使用 `manage.py dbshell`)。 执行 `ALTER TABLE` 语句来添加新列。
4. 使用Python的 `manage.py shell` ， 通过导入模型和选中表单(例如, `MyModel.objects.all()[:5]`)来验证新的字段是否被正确的添加 ,如果一切顺利,所有的语句都不会报错。

然后在你的产品服务器上再实施一遍这些步骤。

1. 启动数据库的交互界面。
2. 执行在开发环境步骤中，第三步的 `ALTER TABLE` 语句。
3. 将新的字段加入到模型中。 如果你使用了某种版本控制工具，并且在第一步中，已经提交了你在开发环境上的修改，现在，可以在生产环境中更新你的代码了（例如，如果你使用Subversion，执行 `svn update` 。
4. 重新启动Web server，使修改生效。

让我们实践下，比如添加一个num_pages字段到第五章中Book模型。首先，我们会把开发环境中的模型改成如下形式：

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
    **num_pages = models.IntegerField(blank=True, null=True)**

    def __unicode__(self):
        return self.title
```

(注意 阅读第六章的“设置可选字段”以及本章下面的“添加非空列”小节以了解我们在这里添加 `blank=True` 和 `null=True` 的原因。)

然后，我们运行命令manage.py sqlall books 来查看CREATE TABLE语句。 语句的具体内容取决与你所使用的数据库， 大概是这个样子：

```
CREATE TABLE "books_book" (
    "id" serial NOT NULL PRIMARY KEY,
    "title" varchar(100) NOT NULL,
    "publisher_id" integer NOT NULL REFERENCES "books_publisher" ("id"),
    "publication_date" date NOT NULL,
    "num_pages" integer NULL
);
```

新加的字段被这样表示：

```
"num_pages" integer NULL
```

接下来，我们要在开发环境上运行数据库客户端，如果是PostgreSQL，运行 psql,，然后，我执行如下语句。

```
ALTER TABLE books_book ADD COLUMN num_pages integer;
```

添加 非NULL 字段

这里有个微妙之处值得一提。 在我们添加字段num_pages的时候，我们使用了 blank=True 和 null=True 选项。 这是因为在我们第一次创建它的时候，这个数据库字段会含有空值。

然而，想要添加不能含有空值的字段也是可以的。 要想实现这样的效果，你必须先创建 NULL 型的字段，然后将该字段的值填充为某个默认值，然后再将该字段改为 NOT NULL 型。 例如：

```
BEGIN;
ALTER TABLE books_book ADD COLUMN num_pages integer;
UPDATE books_book SET num_pages=0;
ALTER TABLE books_book ALTER COLUMN num_pages SET NOT NULL;
COMMIT;
```

如果你这样做，记得你不要在模型中添加 blank=True 和 null=True 选项。

执行ALTER TABLE之后，我们要验证一下修改结果是否正确。 启动python并执行下面的代码：

```
>>> from mysite.books.models import Book
>>> Book.objects.all()[:5]
```

如果没有异常发生，我们将切换到生产服务器，然后在生产环境的数据库中执行命令 `ALTER TABLE` 然后我们更新生产环境中的模型，最后重启web服务器。

删除字段

从Model中删除一个字段要比添加容易得多。 删除字段，仅仅只要以下几个步骤：

> 删除字段，然后重新启动你的web服务器。
>
> 用以下命令从数据库中删除字段：

```
ALTER TABLE books_book DROP COLUMN num_pages;
```

请保证操作的顺序正确。 如果你先从数据库中删除字段，Django将会立即抛出异常。

删除多对多关联字段

由于多对多关联字段不同于普通字段，所以删除操作是不同的。

> 从你的模型中删除 `ManyToManyField`，然后重启web服务器。
>
> 用下面的命令从数据库删除关联表：

```
DROP TABLE books_book_authors;
```

像上面一样，注意操作的顺序。

删除模型

删除整个模型要比删除一个字段容易。 删除一个模型只要以下几个步骤：

> 从文件中删除你想要删除的模型，然后重启web 服务器`models.py`
>
> 然后用以下命令从数据库中删除表：

```
DROP TABLE books_book;
```

> 当你需要从数据库中删除任何有依赖的表时要注意（也就是任何与表books_book有外键的表 ）。

正如在前面部分，一定要按这样的顺序做。

Managers

在语句`Book.objects.all()`中，`objects`是一个特殊的属性，需要通过它查询数据库。 在第5章，我们只是简要地说这是模块的manager 。现在是时候深入了解managers是什么和如何使用了。

总之，模块manager是一个对象，Django模块通过它进行数据库查询。 每个Django模块至少有一个manager，你可以创建自定义manager以定制数据库访问。

下面是你创建自定义manager的两个原因： 增加额外的manager方法，和/或修manager返回的初始QuerySet。

增加额外的**Manager**方法

增加额外的manager方法是为模块添加表级功能的首选办法。 （至于行级功能，也就是只作用于模型对象实例的函数，一会儿将在本章后面解释。）

例如,我们为Book模型定义了一个title_count()方法，它需要一个关键字，返回包含这个关键字的书的数量。 （这个例子有点牵强，不过它可以说明managers如何工作。）

```
# models.py

from django.db import models

# ... Author and Publisher models here ...

**class BookManager(models.Manager):**
    **def title_count(self, keyword):**
        **return self.filter(title__icontains=keyword).count()**

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
    num_pages = models.IntegerField(blank=True, null=True)
    **objects = BookManager()**

    def __unicode__(self):
        return self.title
```

有了这个manager，我们现在可以这样做：

```
>>> Book.objects.title_count('django')
4
>>> Book.objects.title_count('python')
18
```

下面是编码该注意的一些地方：

- 我们建立了一个BookManager类，它继承了django.db.models.Manager。这个类只有一个title_count()方法，用来做统计。 注意，这个方法使用了self.filter()，此处self指manager本身。

- 我们把BookManager()赋值给模型的objects属性。 它将取代模型的默认manager（`objects`）如果我们没有特别定义，它将会被自动创建。 我们把它命名为objects，这是为了与自动创建的manager保持一致。

为什么我们要添加一个title_count()方法呢?是为了将经常使用的查询进行封装，这样我们就不必重复编码了。

修改初始Manager QuerySets

manager的基本QuerySet返回系统中的所有对象。 例如, `Book.objects.all()` 返回数据库book中的所有书本。

我们可以通过覆盖Manager.get_query_set()方法来重写manager的基本QuerySet。get_query_set()按照你的要求返回一个QuerySet。

例如,下面的模型有 两个 manager。一个返回所有对像， 另一个只返回作者是Roald Dahl的书。

```
from django.db import models

**# First, define the Manager subclass.**
**class DahlBookManager(models.Manager):**
    **def get_query_set(self):**
        **return super(DahlBookManager, self).get_query_set().filter(author='Roald Dahl')

**# Then hook it into the Book model explicitly.**
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
    # ...

    **objects = models.Manager() # The default manager.**
    **dahl_objects = DahlBookManager() # The Dahl-specific manager.**
```

在这个示例模型中,Book.objects.all()返回了数据库中的所有书本,而Book.dahl_objects.all()只返回了一本. 注意我们明确地将objects设置成manager的实例，因为如果我们不这么做，那么唯一可用的manager就将是dah1_objects。

当然,由于get_query_set()返回的是一个QuerySet对象，所以我们可以使用filter()，exclude()和其他一切QuerySet的方法。 像这些语法都是正确的：

```
Book.dahl_objects.all()
Book.dahl_objects.filter(title='Matilda')
Book.dahl_objects.count()
```

这个例子也指出了其他有趣的技术： 在同一个模型中使用多个manager。 只要你愿意，你可以为你的模型添加多个manager()实例。 这是一个为模型添加通用滤器的简单方法。

例如:

```
class MaleManager(models.Manager):
    def get_query_set(self):
        return super(MaleManager, self).get_query_set().filter(sex='M')

class FemaleManager(models.Manager):
    def get_query_set(self):
        return super(FemaleManager, self).get_query_set().filter(sex='F')

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    sex = models.CharField(max_length=1, choices=(('M', 'Male'), ('F', 'Female')))
    people = models.Manager()
    men = MaleManager()
    women = FemaleManager()
```

这个例子允许你执行 `Person.men.all()` ， `Person.women.all()` ， `Person.people.all()` 查询，生成你想要的结果。

如果你使用自定义的 `Manager` 对象，请注意，Django遇到的第一个Manager(以它在模型中被定义的位置为准)会有一个特殊状态。 Django将会把第一个 `Manager` 定义为默认 `Manager` ，Django的许多部分(但是不包括admin应用)将会明确地为模型使用这个 `manager` 。 结论是，你

应该小心地选择你的默认manager。因为覆盖 `get_query_set()` 了，你可能接受到一个无用的返回对像，你必须避免这种情况。

模型方法

为了给你的对像添加一个行级功能，那就定义一个自定义方法。 有鉴于manager经常被用来用一些整表操作（table-wide），模型方法应该只对特殊模型实例起作用。

这是一项在模型的一个地方集中业务逻辑的技术。

最好用例子来解释一下。 这个模型有一些自定义方法：

```
from django.contrib.localflavor.us.models import USStateField
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()
    address = models.CharField(max_length=100)
    city = models.CharField(max_length=50)
    state = USStateField() # Yes, this is U.S.-centric...

    def baby_boomer_status(self):
        "Returns the person's baby-boomer status."
        import datetime
        if datetime.date(1945, 8, 1) <= self.birth_date <= datetime.date(1964, 12, 31):
            return "Baby boomer"
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        return "Post-boomer"

    def is_midwestern(self):
        "Returns True if this person is from the Midwest."
        return self.state in ('IL', 'WI', 'MI', 'IN', 'OH', 'IA', 'MO')

    def _get_full_name(self):
        "Returns the person's full name."
        return u'%s %s' % (self.first_name, self.last_name)
    full_name = property(_get_full_name)
```

例子中的最后一个方法是一个property。 想了解更多关于属性的信息请访问 http://www.python.org/download/releases/2.2/descrintro/#property

这是用法的实例：

```
>>> p = Person.objects.get(first_name='Barack', last_name='Obama')
>>> p.birth_date
datetime.date(1961, 8, 4)
>>> p.baby_boomer_status()
'Baby boomer'
>>> p.is_midwestern()
True
>>> p.full_name  # Note this isn't a method -- it's treated as an attribute
u'Barack Obama'
```

执行原始SQL查询

有时候你会发现Django数据库API带给你的也只有这么多，那你可以为你的数据库写一些自定义SQL查询。 你可以通过导入 `django.db.connection` 对像来轻松实现，它代表当前数据库连接。 要使用它，需要通过 `connection.cursor()` 得到一个游标对像。 然后，使用 `cursor.execute(sql, [params])` 来执行SQL语句，使用 `cursor.fetchone()` 或者 `cursor.fetchall()` 来返回记录集。 例如:

```
>>> from django.db import connection
>>> cursor = connection.cursor()
>>> cursor.execute("""
...    SELECT DISTINCT first_name
...    FROM people_person
...    WHERE last_name = %s""", ['Lennon'])
>>> row = cursor.fetchone()
>>> print row
['John']
```

`connection` 和 `cursor` 几乎实现了标准Python DB-API，你可以访问 http://www.python.org/peps/pep-0249.html <http://www.python.org/peps/pep-0249.html&gt __来获取更多信息。 如果你对Python DB-API不熟悉，请注意在 `cursor.execute()` 的SQL语句中使用 `“%s”` ，而不要在SQL内直接添加参数。 如果你使用这项技术，数据库基础库将会自动添加引号，同时在必要的情况下转意你的参数。

不要把你的视图代码和django.db.connection语句混杂在一起，把它们放在自定义模型或者自定义manager方法中是个不错的主意。 比如，上面的例子可以被整合成一个自定义manager方法，就像这样：

```
from django.db import connection, models

class PersonManager(models.Manager):
    def first_names(self, last_name):
        cursor = connection.cursor()
        cursor.execute("""
            SELECT DISTINCT first_name
            FROM people_person
            WHERE last_name = %s""", [last_name])
        return [row[0] for row in cursor.fetchone()]

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    objects = PersonManager()
```

然后这样使用:

```
>>> Person.objects.first_names('Lennon')
['John', 'Cynthia']
```

接下来做什么？

在下一章 我们将讲解Django的通用视图框架，使用它创建常见的网站可以节省时间。

第十一章：通用视图

这里需要再次回到本书的主题： 在最坏的情况下， Web 开发是一项无聊而且单调的工作。 到目前为止，我们已经介绍了 Django 怎样在模型和模板的层面上减小开发的单调性，但是 Web 开发在视图的层面上，也经历着这种令人厌倦的事情。

Django的通用视图 可以减少这些痛苦。 它抽象出一些在视图开发中常用的代码和模式，这样就可以在无需编写大量代码的情况下，快速编写出常用的数据视图。 事实上，前面章节中的几乎所有视图的示例都可以在通用视图的帮助下重写。

在第八章简单的向大家介绍了怎样使视图更加的“通用”。 回顾一下，我们会发现一些比较常见的任务，比如显示一系列对象，写一段代码来显示 任何 对象内容。 解决办法就是传递一个额外的参数到URLConf。

Django内建通用视图可以实现如下功能：

- 完成常用的简单任务： 重定向到另一个页面以及渲染一个指定的模板。
- 显示列表和某个特定对象的详细内容页面。 第8章中提到的 `event_list` 和 `entry_list` 视图就是列表视图的一个例子。 一个单一的 event 页面就是我们所说的详细内容页面。
- 呈现基于日期的数据的年/月/日归档页面，关联的详情页面，最新页面。 Django Weblogs (http://www.djangoproject.com/weblog/)的年、月、日的归档就是使用通用视图架构的，就像是典型的新闻报纸归档。

综上所述，这些视图为开发者日常开发中常见的任务提供了易用的接口。

使用通用视图

使用通用视图的方法是在URLconf文件中创建配置字典，然后把这些字典作为URLconf元组的第三个成员。 （对于这个技巧的应用可以参看第八章向视图传递额外选项。）

例如，下面是一个呈现静态“关于”页面的URLconf：

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template

urlpatterns = patterns('',
    (r'^about/$', direct_to_template, {
        'template': 'about.html'
    })
)
```

一眼看上去似乎有点不可思议，不需要编写代码的视图！ 它和第八章中的例子完全一样： `direct_to_template` 视图仅仅是直接从传递过来的额外参数获取信息并用于渲染视图。

因为通用视图都是标准的视图函数，我们可以在我们自己的视图中重用它。 例如，我们扩展about例子，把映射的URL从 `/about//` 修改到一个静态渲染 `about/.html` 。 我们首先修改URL配置以指向新的视图函数：

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template
**from mysite.books.views import about_pages**

urlpatterns = patterns('',
    (r'^about/$', direct_to_template, {
        'template': 'about.html'
    }),
    **(r'^about/(\w+)/$', about_pages),**
)
```

接下来，我们编写 `about_pages` 视图的代码：

```
from django.http import Http404
from django.template import TemplateDoesNotExist
from django.views.generic.simple import direct_to_template

def about_pages(request, page):
    try:
        return direct_to_template(request, template="about/%s.html" % page)
    except TemplateDoesNotExist:
        raise Http404()
```

在这里我们象使用其他函数一样使用 `direct_to_template` 。 因为它返回一个 `HttpResponse` 对象，我们只需要简单的返回它就好了。 这里唯一有点棘手的事情是要处理找不到模板的情况。 我们不希望一个不存在的模板导致一个服务端错误，所以我们捕获TemplateDoesNotExist异常并且返回404错误来作为替代。

这里有没有安全性问题？

眼尖的读者可能已经注意到一个可能的安全漏洞： 我们直接使用从客户端浏览器得到的数据构造模板名称(`template="about/%s.html" % page`)。乍看起来，这像是一个经典的 目录跨越（*directory traversal*） 攻击（详情请看第20章）。 事实真是这样吗？

完全不是。 是的，一个恶意的 `page` 值可以导致目录跨越，但是尽管 `page` 是 从请求的URL中获取的，但并不是所有的值都会被接受。 这就是URL配置的关键所在： 我们使用正则表达式 `\w+` 来从URL里匹配 `page` ，而 `\w` 只接受字符和数字。 因此，任何恶意的字符（例如在这里是点 `.` 和正斜线 `/` ）将在URL解析时被拒绝，根本不会传递给视图函数。

对象的通用视图

`direct_to_template` 毫无疑问是非常有用的，但Django通用视图最有用的地方是呈现数据库中的数据。 因为这个应用实在太普遍了，Django带有很多内建的通用视图来帮助你很容易 地生成对象的列表和明细视图。

让我们先看看其中的一个通用视图： 对象列表视图。 我们使用第五章中的 `Publisher` 来举例：

```
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __unicode__(self):
        return self.name

    class Meta:
        ordering = ['name']
```

要为所有的出版商创建一个列表页面，我们使用下面的URL配置：

```
from django.conf.urls.defaults import *
from django.views.generic import list_detail
from mysite.books.models import Publisher

publisher_info = {
    'queryset': Publisher.objects.all(),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

这就是所要编写的所有Python代码。 当然，我们还需要编写一个模板。 我们可以通过在额外参数字典中包含一个 `template_name` 键来显式地告诉 `object_list` 视图使用哪个模板：

```
from django.conf.urls.defaults import *
from django.views.generic import list_detail
from mysite.books.models import Publisher

publisher_info = {
    'queryset': Publisher.objects.all(),
    **'template_name': 'publisher_list_page.html',**
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

在缺少 `template_name` 的情况下， `object_list` 通用视图将自动使用一个对象名称。 在这个例子中，这个推导出的模板名称将是 `"books/publisher_list.html"` ，其中books部分是定义这个模型的app的名称， publisher部分是这个模型名称的小写。

这个模板将按照 context 中包含的变量 `object_list` 来渲染，这个变量包含所有的书籍对象。 一个非常简单的模板看起来象下面这样：

```
{% extends "base.html" %}

{% block content %}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list %}
            <li>{{ publisher.name }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

(注意，这里我们假定存在一个 `base.html` 模板，它和我们第四章中的一样。)

这就是所有要做的事。 要使用通用视图酷酷的特性只需要修改参数字典并传递给通用视图函数。 附录D是通用视图的完全参考资料；本章接下来的章节将讲到自定义和扩展通用视图的一些方法。

扩展通用视图

毫无疑问，使用通用视图可以充分加快开发速度。 然而，在多数的工程中，也会出现通用视图不能 满足需求的情况。 实际上，刚接触Django的开发者最常见的问题就是怎样使用通用视图来处理更多的情况。

幸运的是，几乎每种情况都有相应的方法来简易地扩展通用视图以处理这些情况。 这时总是使用下面的 这些方法。

制作友好的模板Context

你也许已经注意到范例中的出版商列表模板在变量 `object_list` 里保存所有的书籍。这个方法工作的很好，只是对编写模板的人不太友好。 他们必须知道这里正在处理的是书籍。 更好的变量名应该是 `publisher_list` ，这样变量所代表的内容就显而易见了。

我们可以很容易地像下面这样修改 `template_object_name` 参数的名称：

```
from django.conf.urls.defaults import *
from django.views.generic import list_detail
from mysite.books.models import Publisher

publisher_info = {
    'queryset': Publisher.objects.all(),
    'template_name': 'publisher_list_page.html',
    'template_object_name': 'publisher',
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

在模板中，通用视图会通过在 `template_object_name` 后追加一个 `_list` 的方式来创建一个表示列表项目的变量名。

使用有用的 `template_object_name` 总是个好想法。 你的设计模板的合作伙伴会感谢你的。

添加额外的Context

你常常需要呈现比通用视图提供的更多的额外信息。 例如，考虑一下在每个出版商的详细页面显示所有其他出版商列表。 `object_detail` 通用视图为context提供了出版商信息，但是看起来没有办法在模板中 获取 所有 出版商列表。

这是解决方法： 所有的通用视图都有一个额外的可选参数 `extra_context` 。这个参数是一个字典数据类型，包含要添加到模板的context中的额外的对象。 所以要给视图提供所有出版商的列表，我们就用这样的info字典：

```
publisher_info = {
    'queryset': Publisher.objects.all(),
    'template_object_name': 'publisher',
    **'extra_context': {'book_list': Book.objects.all()}**
}
```

这样就把一个 `{{ book_list }}` 变量放到模板的context中。 这个方法可以用来传递任意数据到通用视图模板中去，非常方便。 这是非常方便的

不过，这里有一个很隐蔽的BUG，不知道你发现了没有？

我们现在来看一下， `extra_context` 里包含数据库查询的问题。 因为在这个例子中，我们把 `Publisher.objects.all()` 放在URLconf中，它只会执行一次（当URLconf第一次加载的时候）。 当你添加或删除出版商，你会发现在重启Web服务器之前，通用视图不会反映出这些修改（有关QuerySet何时被缓存和赋值的更多信息请参考附录C中“缓存与查询集”一节）。

备注

这个问题不适用于通用视图的 `queryset` 参数。 因为Django知道有些特别的 QuerySet 永远不能 被缓存，通用视图在渲染前都做了缓存清除工作。

解决这个问题的办法是在 *extra_context* 中用一个回调（callback）来代替使用一个变量。 任何传递给 `extra_context` 的可调用对象（例如一个函数）都会在每次视图渲染前执行（而不是只执行一次）。 你可以象这样定义一个函数：

```
**def get_books():**
    **return Book.objects.all()**

publisher_info = {
    'queryset': Publisher.objects.all(),
    'template_object_name': 'publisher',
    'extra_context': **{'book_list': get_books}**
}
```

或者你可以使用另一个不是那么清晰但是很简短的方法，事实上 `Publisher.objects.all` 本身就是可以调用的：

```
publisher_info = {
    'queryset': Publisher.objects.all(),
    'template_object_name': 'publisher',
    'extra_context': **{'book_list': Book.objects.all}**
}
```

注意 `Book.objects.all` 后面没有括号；这表示这是一个函数的引用，并没有真正调用它（通用视图将会在渲染时调用它）。

显示对象的子集

现在让我们来仔细看看这个 `queryset` 。 大多数通用视图有一个 `queryset` 参数，这个参数告诉视图要显示对象的集合 （有关QuerySet的解释请看第五章的 “选择对象”章节，详细资料请参看附录B）。

举一个简单的例子，我们打算对书籍列表按出版日期排序，最近的排在最前：

```
book_info = {
    'queryset': Book.objects.order_by('-publication_date'),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    **(r'^books/$', list_detail.object_list, book_info),**
)
```

这是一个相当简单的例子，但是很说明问题。 当然，你通常还想做比重新排序更多的事。 如果你想要呈现某个特定出版商出版的所有书籍列表，你可以使用同样的技术：

```
**apress_books = {**
    **'queryset': Book.objects.filter(publisher__name='Apress Publishing'),**
    **'template_name': 'books/apress_list.html'**
**}**

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    **(r'^books/apress/$', list_detail.object_list, apress_books),**
)
```

注意 在使用一个过滤的 `queryset` 的同时，我们还使用了一个自定义的模板名称。 如果我们不这么做，通用视图就会用以前的模板，这可能不是我们想要的结果。

同样要注意的是这并不是一个处理出版商相关书籍的最好方法。 如果我们想要添加另一个 出版商页面，我们就得在URL配置中写URL配置，如果有很多的出版商，这个方法就不能 接受了。 在接下来的章节我们将来解决这个问题。

用函数包装来处理复杂的数据过滤

另一个常见的需求是按URL里的关键字来过滤数据对象。 之前，我们在URLconf中硬编码了出版商的名字，但是如果我们想用一个视图就显示某个任意指定的出版商的所有书籍，那该怎么办呢？ 我们可以通过对 `object_list` 通用视图进行包装来避免 写一大堆的手工代码。 按惯例，我们先从写URL配置开始：

```
urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    **(r'^books/(\w+)/$', books_by_publisher),**
)
```

接下来，我们写 `books_by_publisher` 这个视图：

```
from django.shortcuts import get_object_or_404
from django.views.generic import list_detail
from mysite.books.models import Book, Publisher

def books_by_publisher(request, name):

    # Look up the publisher (and raise a 404 if it can't be found).
    publisher = get_object_or_404(Publisher, name__iexact=name)

    # Use the object_list view for the heavy lifting.
    return list_detail.object_list(
        request,
        queryset = Book.objects.filter(publisher=publisher),
        template_name = 'books/books_by_publisher.html',
        template_object_name = 'book',
        extra_context = {'publisher': publisher}
    )
```

这样写没问题，因为通用视图就是Python函数。 和其他的视图函数一样，通用视图也是接受一些 参数并返回 `HttpResponse` 对象。 因此，通过包装通用视图函数可以做更多的事。

注意

注意在前面这个例子中我们在 `extra_context` 中传递了当前出版商这个参数。

处理额外工作

我们再来看看最后一个常用模式：

想象一下我们在 `Author` 对象里有一个 `last_accessed` 字段，我们用这个字段来记录最近一次对author的访问。 当然通用视图 `object_detail` 并不能处理这个问题，但是我们仍然可以很容易地编写一个自定义的视图来更新这个字段。

首先，我们需要在URL配置里设置指向到新的自定义视图：

```
from mysite.books.views import author_detail

urlpatterns = patterns('',
    # ...
    **(r'^authors/(?P<author_id>\d+)/$', author_detail),**
    # ...
)
```

接下来写包装函数：

```
import datetime
from django.shortcuts import get_object_or_404
from django.views.generic import list_detail
from mysite.books.models import Author

def author_detail(request, author_id):
    # Delegate to the generic view and get an HttpResponse.
    response = list_detail.object_detail(
        request,
        queryset = Author.objects.all(),
        object_id = author_id,
    )

    # Record the last accessed date. We do this *after* the call
    # to object_detail(), not before it, so that this won't be called
    # unless the Author actually exists. (If the author doesn't exist,
    # object_detail() will raise Http404, and we won't reach this point.)
    now = datetime.datetime.now()
    Author.objects.filter(id=author_id).update(last_accessed=now)

    return response
```

注意

除非你添加 `last_accessed` 字段到你的 `Author` 模型并创建 `books/author_detail.html` 模板，否则这段代码不能真正工作。

我们可以用同样的方法修改通用视图的返回值。 如果我们想要提供一个供下载用的 纯文本版本的author列表，我们可以用下面这个视图：

```
def author_list_plaintext(request):
    response = list_detail.object_list(
        request,
        queryset = Author.objects.all(),
        mimetype = 'text/plain',
        template_name = 'books/author_list.txt'
    )
    response["Content-Disposition"] = "attachment; filename=authors.txt"
    return response
```

这个方法之所以工作是因为通用视图返回的 `HttpResponse` 对象可以象一个字典 一样的设置HTTP的头部。 随便说一下，这个 `Content-Disposition` 的含义是 告诉浏览器下载并保存这个页面，而不是在浏览器中显示它。

下一章

在这一章我们只讲了Django带的通用视图其中一部分，不过这些方法也适用于其他的 通用视图。 附录C详细地介绍了所有可用的视图，如果你想了解这些强大的特性，推荐你阅读一下。

这本书的高级语法部分到此结束。 在下一章, 我们讲解了Django应用的部署。

第十二章： 部署Django

本章包含创建一个django程序最必不可少的步骤 在服务器上部署它

如果你一直跟着我们的例子做，你可能正在用 `runserver` 但是 `runserver` 要部署你的django程序，你需要挂接到工业用的服务器 如：Apache 在本章，我们将展示如何做，但是，在做之前我们要给你一个(要做的事的)清单.

准备你的代码库

很幸运， `runserver` 但是，在开始前，有一些**

关闭Debug模式.

我们在第2章,用命令 `django-admin.py startproject` 创建了一个项目 , 其中创建的 `settings.py` 文件的 `DEBUG` 设置默认为 `True` . django会根据这个设置来改变他们的行为，如果 `DEBUG` 模式被开启. 例如， 如果 `DEBUG` 被设置成 `True` , 那么:

- 所有的数据库查询将被保存在内存中， 以 `django.db.connection.queries` 的形式. 你可以想象，这个吃内存!
- 任何404错误都将呈现django的特殊的404页面(第3章有)而不是普通的404页面。 这个页面包含潜在的敏感信息，但是不会暴露在公共互联网。
- 你的应用中任何未捕获的异常，从基本的python语法错误到数据库错误以及模板语法错误都会返回漂亮的Django错误页面。 这个页面包含了比404错误页面更多的敏感信息，所以这个页面绝对不要公开暴露。

简单的说，把 `DEBUG` 设置成 `True` 相当于告诉Django你的网站只会被可信任的开发人员使用。 Internet里充满了不可信赖的事物，当你准备部署你的应用时，首要的事情就是把 `DEBUG` 设置为 `False` 。

来关闭模板Debug模式。

类似地，你应该在生产环境中把 `TEMPLATE_DEBUG``False` 如果这个设为 `True` ， 为了在那个好看的错误页面上显示足够的东西，Django的模版系统就会为每一个模版保存一些额外的信息。

实现一个404模板

如果 `DEBUG` 设置为 `True` ，Django会显示那个自带的404错误页面。 但如果 `DEBUG` 被设置成 `False` ，那它的行为就不一样了： 他会显示一个在你的模版根目录中名字叫 `404.html` 的模版 所以，当你准备部署你的应用时，你会需要创建这个模版并在里面放一些有意义的“页面未找到”的信息

这里有一个 `404.html` 的示例，你可以从它开始。 假定你使用的模板继承并定义一个 `base.html` ,该页面由 `title``content` 两块组成。

```
{% extends "base.html" %}

{% block title %}Page not found{% endblock %}

{% block content %}
<h1>Page not found</h1>

<p>Sorry, but the requested page could not be found.</p>
{% endblock %}
```

要测试你的 `404.html` 页面是否正常工作，仅仅需要将 `DEBUG` 设置为 `False` ，并且访问一个并不存在的URL。 （它将在 `sunserver` 上工作的和开发服务器上一样好）

实现一个500模板

类似的，如果 `DEBUG` 设置为 `False` ，Djang不再会显示它自带的应对未处理的Python异常的错误反馈页面。 作为代替，它会查找一个名为 `500.html` 的模板并且显示它。 像 `404.html` 一样，这个模板应该被放置在你的模板根目录下。

这里有一个关于500.html的比较棘手的问题。你永远不能确定 `为什么` 会显示这个模板，所以它不应该做任何需要连接数据库，或者依赖任何可能被破坏的基础构件的事情。 （例如：它不应该使用自定义模板标签。）如果它用到了模板继承，那么父模板也就不应该依赖可能被破坏的基础构件。 因此，最好的方法就是避免模板继承，并且用一些非常简单的东西。 这是一个 `500.html` 的例子，可以把它作为一个起点：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
    <title>Page unavailable</title>
</head>
<body>
    <h1>Page unavailable</h1>

    <p>Sorry, but the requested page is unavailable due to a
    server hiccup.</p>

    <p>Our engineers have been notified, so check back later.</p>
</body>
</html>
```

设置错误警告

当你使用Django制作的网站运行中出现了异常，你会希望去了解以便于修正它。 默认情况下，Django在你的代码引发未处理的异常时，将会发送一封Email至开发者团队。但你需要去做两件事来设置这种行为。

首先，改变你的 `ADMINS` 设置用来引入你的E-mail地址，以及那些任何需要被注意的联系人的E-mail地址。 这个设置采用了类似于 `(姓名, Email)` 元组，像这样：

```
ADMINS = (
    ('John Lennon', 'jlennon@example.com'),
    ('Paul McCartney', 'pmacca@example.com'),
)
```

第二，确保你的服务器配置为发送电子邮件。 设置好postfix,sendmail或其他本书范围之外但是与Django设置相关的邮件服务器,你需要将将 EMAIL_HOST设置为你的邮件服务器的正确的主机名. 默认模式下是设置为'localhost', 这个设置对大多数的共享主机系统环境适用. 取决于你的安排的复杂性,你可能还需要设置EMAIL_HOST_USER,EMAIL_HOST_PASSWORD,EMAIL_PORT或EMAIL_USE_TLS。

你还可以设置EMAIL_SUBJECT_PREFIX以控制Django使用的 error e-mail的前缀。 默认情况下它被设置为 `'[Django] '`

设置连接中断警报

如果你安装有CommonMiddleware(比如，你的MIDDLEWARE_CLASSES设置包含了'django.middleware.common.CommonMiddleware'的情况下，默认就安装了CommonMiddleware),你就具有了设置这个选项的能力：有人在访问你的Django网站的一个非空的链接而导致一个404错误的发生和连接中断的情况，你将收到一封邮件. 如果你想激活这个特性，设置SEND_BROKEN_LINK_EMAILS 为True(默认为False),并设置你的MANAGERS为某个人或某些人的邮件地址，这些邮件地址将会收到报告连接中断错误的邮件. MANAGERS使用和ADMINS 同样的语法.例如:

```
MANAGERS = (
    ('George Harrison', 'gharrison@example.com'),
    ('Ringo Starr', 'ringo@example.com'),
)
```

请注意，错误的Email会令人感到反感，对于任何人来说都是这样。

使用针对产品的不同的设置

在此书中，我们仅仅处理一个单一的设置文件 settings.py文件由django-admin.py startproject命令生成。但是当你准备要进行配置的时候，你将发现你需要多个配置文件以使你的开发环境和产品环境相独立。 比如，你可能不想每次在本地机器上测试代码改变的时候将DEBUG从False 改为True。Django通过使用多个配置文件而使得这种情况很容易得到避免。

如果你想把你的配置文件按照产品设置和开发设置组织起来，你可以通过下面三种方法的其中一种达到这个目的。

- 设置成两个全面的，彼此独立的配置文件
- 设置一个基本的配置文件（比如，为了开发)和第二个（为了产品)配置文件，第二个配置文件仅仅从基本的那个配置文件导入配置，并对需要定义的进行复写.
- 使用一个单独的配置文件，此配置文件包含一个Python的逻辑判断根据上下文环境改变设置。

我们将会在依次解释这几种方式

首先，最基本的方法是定义两个单独的配置文件。 如果你是跟随之前的例子做下来的，那么你已经有了一个settings.py了，现在你只需要将它复制一份并命名为settings_production.py（文件名可以按照你自己的喜好定义),在这个新文件中改变DEBUG等设置。

第二种方法比较类似，但是减少了许多冗余。 作为使用两个内容大部分相同的配置文件的替代方式，你可以使用一个文件为基本文件，另外一个文件从基本文件中导入相关设定。 例如

```
# settings.py

DEBUG = True
TEMPLATE_DEBUG = DEBUG

DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_NAME = 'devdb'
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_PORT = ''

# ...

# settings_production.py

from settings import *

DEBUG = TEMPLATE_DEBUG = False
DATABASE_NAME = 'production'
DATABASE_USER = 'app'
DATABASE_PASSWORD = 'letmein'
```

此处，settings_production.py 从settings.py 导入所有的设定，仅仅只是重新定义了产品模式下需要特殊处理的设置。 在这个案例中，DEBUG 被设置为False，但是我们已经对产品模式设置了不同的数据库访问参数。 （后者将向你演示你可以重新定义 任何 设置，并不只是象DEBUG 这样的基本设置。）

最终，最精简的达到两个配置环境设定的方案是使用一个配置文件，在此配置文件中根据不同的环境进行设置。 一个达到这个目的的方法是检查当前的主机名。 例如：

```
# settings.py

import socket

if socket.gethostname() == 'my-laptop':
    DEBUG = TEMPLATE_DEBUG = True
else:
    DEBUG = TEMPLATE_DEBUG = False

# ...
```

在这里，我们从python标准库导入了socket 模块，使用它来检查当前系统的主机名。 我们可以通过检查主机名来确认代码是否运行在产品服务器上。

一个关键是配置文件仅仅是包含python代码的文件。你可以从其他文件导入这些python代码，可以通过这些代码执行任意的逻辑判断等操作。 如果你打算按照这种方案走下去，请确定这些配置文件中的代码是足够安全（防弹)的。 如果这个配置文件抛出任何的异常，Django都有可能会发生很严重的崩溃。

重命名settings.py

随便将你的settings.py重命名为settings_dev.py或settings/dev.py或foobar.py，Django 并不在乎你的配置文件取什么名字，只要你告诉它你使用的哪个配置文件就可以了。

但是如果你真的重命名了由django-admin.py startproject 命令创建的settings.py文件，你会发现manage.py会给出一个错误信息说找不到配置文件。 那是由于它尝试从这个文件中导入一个叫做settings的模块，你可以通过修改manage.py 文件，将 import settings 语句改为导入你自己的模块，或者使用django-admin.py而不是使用manage.py,在后一种方式中你需要设置 DJANGO_SETTINGS_MODULE 环境变量为你的配置文件所在的python 路径.(比如'mysite.settings'）。

DJANGO_SETTINGS_MODULE

通过这种方式的代码改变后，本章的下一部分将集中在对具体环境(比如Apache)的发布所需要的指令上。 这些指令针对每一种环境都不同，但是有一件事情是相同的。 在每一种环境中，你都需要告诉Web服务器你的DJANGO_SETTINGS_MODULE是什么,这是你的Django应用程序的进入点。 DJANGO_SETTINGS_MODULE指向你的配置文件，在你的配置文件中指向你的ROOT_URLCONF,在ROOT_URLCONF中指向了你的视图以及其他的部分。

DJANGO_SETTINGS_MODULE是你的配置文件的python的路径 比如，假设mysite是在你的Python路径中，DJANGO_SETTINGS_MODULE对于我们正在进行的例子就是'mysite.settings'。

用Apache和mod_python来部署Django

目前，Apache和mod_python是在生产服务器上部署Django的最健壮搭配。

mod_python (http://www.djangoproject.com/r/mod_python/)是一个在Apache中嵌入Python的Apache插件，它在服务器启动时将Python代码加载到内存中。 (译注：

Django 需要Apaceh 2.x 和mod_python 3.x支持。

备注

如何配置Apache超出了本书的范围，因此下面将只简单介绍必要的细节。 幸运的是，如果需要进一步学习Apache的相关知识，可以找到相当多的绝佳资源。 我们喜欢去的几个地方:

- 开源的Apache在线文档，位于 http://www.djangoproject.com/r/apache/docs/
- *Pro Apache*，第三版 (Apress, 2004),作者Peter Wainwright, 位于 http://www.djangoproject.com/r/books/pro-apache/
- *Apache: The Definitive Guide,* 第三版 (OReilly, 2002),作者Ben Laurie和Peter Laurie, 位于 http://www.djangoproject.com/r/books/apache-pra/

基本配置

为了配置基于 mod_python 的 Django，首先要安装有可用的 mod_python 模块的 Apache。 这通常意味着应该有一个 `LoadModule` 指令在 Apache 配置文件中。 它看起来就像是这样：

```
LoadModule python_module /usr/lib/apache2/modules/mod_python.so
```

Then, edit your Apache configuration file and add a `<Location>` directive that ties a specific URL path to a specific Django installation. 例如：

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug Off
</Location>
```

要确保把 `DJANGO_SETTINGS_MODULE` 中的 `mysite.settings` 项目换成与你的站点相应的内容。

它告诉 Apache，任何在 / 这个路径之后的 URL 都使用 Django 的 mod_python 来处理。 它将 `DJANGO_SETTINGS_MODULE` 的值传递过去，使得 mod_python 知道这时应该使用哪个配置。

注意这里使用 `` `指令而不是 [` ](#id11) 。 `后者用于指向你的文件系统中的一个位置，然而 [` ](#id13)``

System Message: WARNING/2 (`<string>` , line 403); *backlink*

Inline literal start-string without end-string.

System Message: WARNING/2 (`<string>` , line 403); *backlink*

Inline literal start-string without end-string.

System Message: WARNING/2 (`<string>` , line 403); *backlink*

Inline literal start-string without end-string.

System Message: WARNING/2 (`<string>` , line 403); *backlink*

Inline literal start-string without end-string.

System Message: ERROR/3 (`<string>` , line 405)

Unexpected indentation.

> 指向一个 Web 站点的 URL 位置。 ````
>
> System Message: WARNING/2 (`<string>` , line 405); *backlink*
>
> Inline literal start-string without end-string.
>
> System Message: WARNING/2 (`<string>` , line 405); *backlink*
>
> Inline literal start-string without end-string.

Apache 可能不但会运行在你正常登录的环境中，也会运行在其它不同的用户环境中；也可能会有不同的文件路径或 sys.path。 你需要告诉 mod_python 如何去寻找你的项目及 Django 的位置。

```
PythonPath "['/path/to/project', '/path/to/django'] + sys.path"
```

你也可以加入一些其它指令，比如 `PythonAutoReload Off` 以提升性能。 查看 mod_python 文档获得详细的指令列表。

注意，你应该在成品服务器上设置 `PythonDebug Off` 。如果你使用 `PythonDebug On` 的话，在程序产生错误时，你的用户会看到难看的（并且是暴露的） Python 回溯信息。 如果你把 PythonDebug 置 On,当mod_python出现某些错误,你的用户会看到丑陋的（也会暴露某些信息)Python的对错误的追踪的信息。

重启 Apache 之后所有对你的站点的请求（或者是当你用了 `<VirtualHost>` 指令后则是虚拟主机）都会由 Djanog 来处理。

在同一个 Apache 的实例中运行多个 Django 程序

在同一个 Apache 实例中运行多个 Django 程序是完全可能的。 当你是一个独立的 Web 开发人员并有多个不同的客户时，你可能会想这么做。

只要像下面这样使用 `VirtualHost` 你可以实现：

```
NameVirtualHost *

<VirtualHost *>
    ServerName www.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</VirtualHost>

<VirtualHost *>
    ServerName www2.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
</VirtualHost>
```

如果你需要在同一个 `VirtualHost` 中运行两个 Django 程序，你需要特别留意一下以 确保 mod_python 的代码缓存不被弄得乱七八糟。 使用 `PythonInterpreter` 指令来将不 同的 `<Location>` 指令分别解释：

```
<VirtualHost *>
    ServerName www.example.com
    # ...
    <Location "/something">
        SetEnv DJANGO_SETTINGS_MODULE mysite.settings
        PythonInterpreter mysite
    </Location>

    <Location "/otherthing">
        SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
        PythonInterpreter mysite_other
    </Location>
</VirtualHost>
```

这个 `PythonInterpreter` 中的值不重要，只要它们在两个 `Location` 块中不同。

用 **mod_python** 运行一个开发服务器

因为 mod_python 缓存预载入了 Python 的代码，当在 mod_python 上发布 Django 站点时，你每 改动了一次代码都要需要重启 Apache 一次。 这还真是件麻烦事，所以这有个办法来避免它： 只要 加入 `MaxRequestsPerChild 1` 到配置文件中强制 Apache 在每个请求时都重新载入所有的 代码。 但是不要在产品服务器上使用这个指令，这会撤销 Django 的特权。

如果你是一个用分散的 `print` 语句（我们就是这样）来调试的程序员，注意这 `print` 语 句在 mod_python 中是无效的；它不会像你希望的那样产生一个 Apache 日志。 如果你需要在 mod_python 中打印调试信息，可能需要用到 Python 标准日志包（Pythons standard logging package）。 更多的信息请参见 http://docs.python.org/lib/module-logging.html 。另一个选择是在模板页面中加入调试信息。

使用相同的**Apache**实例来服务**Django**和**Media**文件

Django本身不用来服务media文件；应该把这项工作留给你选择的网络服务器。 我们推荐使用一个单独的网络服务器（即没有运行Django的一个）来服务media。 想了解更多信息，看下面的章节。

不过，如果你没有其他选择，所以只能在同Django一样的Apache `VirtualHost` 上服务media文件，这里你可以针对这个站点的特定部分关闭mod_python：

```
<Location "/media/">
    SetHandler None
</Location>
```

将 `Location` 改成你的media文件所处的根目录。

你也可以使用 `<LocationMatch>` 来匹配正则表达式。 比如，下面的写法将Django定义到网站的根目录，并且显式地将 `media` 子目录以及任何以 `.jpg` ， `.gif` ， 或者 `.png` 结尾的URL屏蔽掉:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</Location>

<Location "/media/">
    SetHandler None
</Location>

<LocationMatch "\.(jpg|gif|png)$">
    SetHandler None
</LocationMatch>
```

在所有这些例子中，你必须设置 `DocumentRoot` ，这样apache才能知道你存放静态文件的位置。

错误处理

当你使用 Apache/mod_python 时，错误会被 Django 捕捉，它们不会传播到 Apache 那里，也不会出现在 Apache 的 `错误日志` 中。

除非你的 Django 设置的确出了问题。 在这种情况下，你会在浏览器上看到一个 内部服务器错误的页面，并在 Apache 的 `错误日志` 中看到 Python 的完整回溯信息。 `错误日志` 的回溯信息有多行。 当然，这些信息是难看且难以阅读的。

处理段错误

有时候，Apache会在你安装Django的时候发生段错误。 这时，基本上 总是 有以下两个与Django本身无关的原因其中之一所造成：

- 有可能是因为，你使用了 `pyexpat` 模块（进行XML解析）并且与Apache内置的版本相冲突。 详情请见 http://www.djangoproject.com/r/articles/expat-apache-crash/.
- 也有可能是在同一个Apache进程中，同时使用了mod_python 和 mod_php，而且都使用MySQL作为数据库后端。 在有些情况下，这会造成PHP和Python的MySQL模块的版本冲突。 在mod_python的FAQ中有更详细的解释。

如果还有安装mod_python的问题，有一个好的建议，就是先只运行mod_python站点，而不使用Django框架。 这是区分mod_python特定问题的好方法。 下面的这篇文章给出了更详细的解释。 http://www.djangoproject.com/r/articles/getting-modpython-working/.

下一个步骤应该是编辑一段测试代码，把你所有django相关代码import进去， 你的views,models,URLconf,RSS配置， 等等。 把这些imports放进你的handler函数中， 然后从浏览器进入你的URL。 如果这些导致了crash，你就可以确定是import的django代码引起了问题。 逐个去掉这些imports，直到不再冲突，这样就能找到引起问题的那个模块。 深入了解各模块，看看它们的imports。 要想获得更多帮助，像linux的ldconfig，Mac OS的otool和windows的ListDLLs（form sysInternals）都可以帮你识别共享依赖和可能的版本冲突。

一种替代方案： mod_wsgi模块

作为一个mod_python模块的替代，你可以考虑使用mod_wsgi模块(http://code.google.com/p/modwsgi/),此模块开发的时间比mod_python的开发时间离现在更近一些，在Django社区已有一些使用。 一个完整的概述超出了本书的范围，你可以从官方的Django文档查看到更多的信息。

使用FastCGI部署Django应用

尽管将使用Apache和mod_python搭建Django环境是最具鲁棒性的，但在很多虚拟主机平台上，往往只能使用FastCGI

此外，在很多情况下，FastCGI能够提供比mod_python更为优越的安全性和效能。 针对小型站点，相对于Apache来说FastCGI更为轻量级。

FastCGI 简介

如何能够由一个外部的应用程序很有效解释WEB 服务器上的动态页面请求呢？ 答案就是使用FastCGI! 它的工作步骤简单的描述起来是这样的：

和mod_python一样，FastCGI也是驻留在内存里为客户请求返回动态信息,而且也免掉了像传统的CGI一样启动进程时候的时间花销。 但于mod_python不同之处是它并不是作为模块运行在web服务器同一进程内的，而是有自己的独立进程。

为什么要在一个独立的进程中运行代码？

在以传统的方式的几种以mod_*方式嵌入到Apache的脚本语言中（常见的例如：PHP，Python/mod_python和Perl/mod_perl），他们都是以apache扩展模块的方式将自身嵌入到Apache进程中的。

每一个Apache进程都是一个Apache引擎的副本，它完全包括了所有Apache所具有的一切功能特性（哪怕是对Django毫无好处的东西也一并加载进来）。而FastCGI就不一样了，它仅仅把Python和Django等必备的东东弄到内存中。

依据FastCGI自身的特点可以看到，FastCGI进程可以与Web服务器的进程分别运行在不同的用户权限下。对于一个多人共用的系统来说，这个特性对于安全性是非常有好处的，因为你可以安全的于别人分享和重用代码了。

如果你希望你的Django以FastCGI的方式运行，那么你还必须安装 `flup` 这个Python库，这个库就是用于处理FastCGI的。很多用户都抱怨 `flup` 的发布版太久了，老是不更新。其实不是的，他们一直在努力的工作着，这是没有放出来而已。

运行你的 FastCGI 服务器

FastCGI是以客户机/服务器方式运行的，并且在很多情况下，你得自己去启动FastCGI的服务进程。Web服务器（例如Apache,lighttpd等等）仅仅在有动态页面访问请求的时候才会去与你的Django-FastCGI进程交互。因为Fast-CGI已经一直驻留在内存里面了的，所以它响应起来也是很快的。

记录

在虚拟主机上使用的话，你可能会被强制的使用Web server-managed FastCGI进程。在这样的情况下，请参阅下面的“在Apache共享主机里运行Django”这一小节。

web服务器有两种方式于FastCGI进程交互：使用Unix domain socket(在win32里面是 命名管道)或者使用TCP socket.具体使用哪一个，那就根据你的偏好而定了，但是TCP socket弄不好的话往往会发生一些权限上的问题。What you choose is a manner of preference; a TCP socket is usually easier due to permissions issues.

开始你的服务器项目，首先进入你的项目目录下（你的 `manage.py` 文件所在之处），然后使用 `manage.py runfcgi` 命令：

```
./manage.py runfcgi [options]
```

想了解如何使用 `runfcgi` ，输入 `manage.py runfcgi help` 命令。

你可以指定 `socket` 或者同时指定 `host` 和 `port` 。当你要创建Web服务器时，你只需要将服务器指向当你在启动FastCGI服务器时确定的socket或者host/port。

范例：

> 在TCP端口上运行一个线程服务器：

```
./manage.py runfcgi method=threaded host=127.0.0.1 port=3033
```

> 在Unix socket上运行prefork服务器：

```
./manage.py runfcgi method=prefork socket=/home/user/mysite.sock pidfile=django.pid
```

> 启动，但不作为后台进程（在调试时比较方便）：

```
./manage.py runfcgi daemonize=false socket=/tmp/mysite.sock
```

停止FastCGI的行程

如果你的FastCGI是在前台运行的，那么只需按Ctrl+C就可以很方便的停止这个进程了。 但如果是在后台运行的话，你就要使用Unix的 `kill` 命令来杀掉它。 然而，当你正在处理后台进程时，你会需要将其付诸于Unix `kill` 的命令

如果你在 `manage.py runfcgi` 中指定了 `pidfile` 这个选项，那么你可以这样来杀死这个FastCGI后台进程：

```
kill `cat $PIDFILE`
```

`$PIDFILE` 就是你在 `pidfile` 指定的那个。

你可以使用下面这个脚本方便地重启Unix里的FastCGI守护进程：

```
#!/bin/bash

# Replace these three settings.
PROJDIR="/home/user/myproject"
PIDFILE="$PROJDIR/mysite.pid"
SOCKET="$PROJDIR/mysite.sock"

cd $PROJDIR
if [ -f $PIDFILE ]; then
    kill `cat -- $PIDFILE`
    rm -f -- $PIDFILE
fi

exec /usr/bin/env -   PYTHONPATH="../python:.."   ./manage.py runfcgi socket=$SOCKET pidf
```

在Apache中以FastCGI的方式使用Django

在Apache和FastCGI上使用Django，你需要安装和配置Apache，并且安装mod_fastcgi。 请参见Apache和mod_fastcgi文档： http://www.djangoproject.com/r/mod_fastcgi/ 。

当完成了安装，通过 `httpd.conf` （Apache的配置文件）来让Apache和Django FastCGI互相通信。 你需要做两件事：

- 使用 `FastCGIExternalServer` 指明FastCGI的位置。
- 使用 `mod_rewrite` 为FastCGI指定合适的URL。

指定 FastCGI Server 的位置

`FastCGIExternalServer` 告诉Apache如何找到FastCGI服务器。 按照FastCGIExternalServer 文档（ http://www.djangoproject.com/r/mod_fastcgi/FastCGIExternalServer/ ），你可以指明 `socket` 或者 `host` 。以下是两个例子：

```
# Connect to FastCGI via a socket/named pipe:
FastCGIExternalServer /home/user/public_html/mysite.fcgi -socket /home/user/mysite.sock

# Connect to FastCGI via a TCP host/port:
FastCGIExternalServer /home/user/public_html/mysite.fcgi -host 127.0.0.1:3033
```

在这两个例子中， /home/user/public_html/ 目录必须存在，而 `/home/user/public_html/mysite.fcgi` 文件不一定存在。 它仅仅是一个Web服务器内部使用的接口，这个URL决定了对于哪些URL的请求会被FastCGI处理（下一部分详细讨论）。 （下一章将会有更多有关于此的介绍）

使用mod_rewrite为FastCGI指定URL

第二步是告诉Apache为符合一定模式的URL使用FastCGI。 为了实现这一点，请使用 mod_rewrite 模块，并将这些URL重定向到 `mysite.fcgi` （或者正如在前文中描述的那样，使用任何在 `FastCGIExternalServer` 指定的内容）。

在这个例子里面，我们告诉Apache使用FastCGI来处理那些在文件系统上不提供文件(译者注：

```
<VirtualHost 12.34.56.78>
  ServerName example.com
  DocumentRoot /home/user/public_html
  Alias /media /home/user/python/django/contrib/admin/media
  RewriteEngine On
  RewriteRule ^/(media.*)$ /$1 [QSA,L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^/(.*)$ /mysite.fcgi/$1 [QSA,L]
</VirtualHost>
```

FastCGI 和 lighttpd

lighttpd (http://www.djangoproject.com/r/lighttpd/) 是一个轻量级的Web服务器，通常被用来提供静态页面的访问。 它天生支持FastCGI，因此除非你的站点需要一些Apache特有的特性，否则，lighttpd对于静态和动态页面来说都是理想的选择。

确保 `mod_fastcgi` 在模块列表中，它需要出现在 `mod_rewrite` 和 `mod_access` ，但是要在 `mod_accesslog` 之前。

将下面的内容添加到你的lighttpd的配置文件中：

```
server.document-root = "/home/user/public_html"
fastcgi.server = (
    "/mysite.fcgi" => (
        "main" => (
            # Use host / port instead of socket for TCP fastcgi
            # "host" => "127.0.0.1",
            # "port" => 3033,
            "socket" => "/home/user/mysite.sock",
            "check-local" => "disable",
        )
    ),
)
alias.url = (
    "/media/" => "/home/user/django/contrib/admin/media/",
)

url.rewrite-once = (
    "^(/media.*)$" => "$1",
    "^/favicon\.ico$" => "/media/favicon.ico",
    "^(/.*)$" => "/mysite.fcgi$1",
)
```

在一个lighttpd进程中运行多个Django站点

lighttpd允许你使用条件配置来为每个站点分别提供设置。 为了支持FastCGI的多站点，只需要在FastCGI的配置文件中，为每个站点分别建立条件配置项：

```
# If the hostname is 'www.example1.com'...
$HTTP["host"] == "www.example1.com" {
    server.document-root = "/foo/site1"
    fastcgi.server = (
       ...
    )
    ...
}

# If the hostname is 'www.example2.com'...
$HTTP["host"] == "www.example2.com" {
    server.document-root = "/foo/site2"
    fastcgi.server = (
       ...
    )
    ...
}
```

你也可以通过 `fastcgi.server` 中指定多个入口，在同一个站点上实现多个Django安装。 请为每一个安装指定一个FastCGI主机。

在使用Apache的共享主机服务商处运行Django

许多共享主机的服务提供商不允许运行你自己的服务进程，也不允许修改 `httpd.conf` 文件。 尽管如此，仍然有可能通过Web服务器产生的子进程来运行Django。

记录

如果你要使用服务器的子进程，你没有必要自己去启动FastCGI服务器。 Apache会自动产生一些子进程，产生的数量按照需求和配置会有所不同。

在你的Web根目录下，将下面的内容增加到 `.htaccess` 文件中：

```
AddHandler fastcgi-script .fcgi
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ mysite.fcgi/$1 [QSA,L]
```

接着，创建一个脚本，告知Apache如何运行你的FastCGI程序。 创建一个 `mysite.fcgi` 文件，并把它放在你的Web目录中，打开可执行权限。

```
#!/usr/bin/python
import sys, os

# Add a custom Python path.
sys.path.insert(0, "/home/user/python")

# Switch to the directory of your project. (Optional.)
# os.chdir("/home/user/myproject")

# Set the DJANGO_SETTINGS_MODULE environment variable.
os.environ['DJANGO_SETTINGS_MODULE'] = "myproject.settings"

from django.core.servers.fastcgi import runfastcgi
runfastcgi(method="threaded", daemonize="false")
```

重启新产生的进程服务器

如果你改变了站点上任何的python代码，你需要告知FastCGI。 但是，这不需要重启Apache，而只需要重新上传 `mysite.fcgi` 或者编辑改文件，使得修改时间发生了变化，它会自动帮你重启Django应用。 你可以重新上传mysite.fcgi或者编辑这个文件以改变该文件的时间戳。 当阿帕奇服务器发现文档被更新了，它将会为你重启你的Django应用。

如果你拥有Unix系统命令行的可执行权限，只需要简单地使用 `touch` 命令：

```
touch mysite.fcgi
```

可扩展性

既然你已经知道如何在一台服务器上运行Django，让我们来研究一下，如何扩展我们的Django安装。 这一部分我们将讨论，如何把一台服务器扩展为一个大规模的服务器集群，这样就能满足每小时上百万的点击率。

有一点很重要，每一个大型的站点大的形式和规模不同，因此可扩展性其实并不是一种千篇一律的行为。 以下部分会涉及到一些通用的原则，并且会指出一些不同选择。

首先，我们来做一个大的假设，只集中地讨论在Apache和mod_python下的可扩展性问题。尽管我们也知道一些成功的中型和大型的FastCGI策略，但是我们更加熟悉Apache。

运行在一台单机服务器上

大多数的站点一开始都运行在单机服务器上，看起来像图20-1这样的构架。

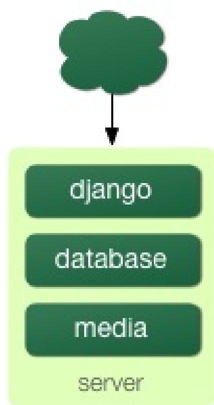


图 20-1： 一个单服务器的Django安装。

这对于小型和中型的站点来说还不错，并且也很便宜，一般来说，你可以在3000美元以下就搞定一切。

然而，当流量增加的时候，你会迅速陷入不同软件的 资源争夺 之中。 数据库服务器和Web服务器都 喜欢 自己拥有整个服务器资源，因此当被安装在单机上时，它们总会争夺相同的资源（RAM, CPU），它们更愿意独享资源。

通过把数据库服务器搬移到第二台主机上，可以很容易地解决这个问题。

分离出数据库服务器

对于Django来说，把数据库服务器分离开来很容易： 只需要简单地修改 `DATABASE_HOST` ，设置为新的数据库服务器的IP地址或者DNS域名。 设置为IP地址总是一个好主意，因为使用DNS域名，还要牵涉到DNS服务器的可靠性连接问题。

使用了一个独立的数据库服务器以后，我们的构架变成了图20-2。

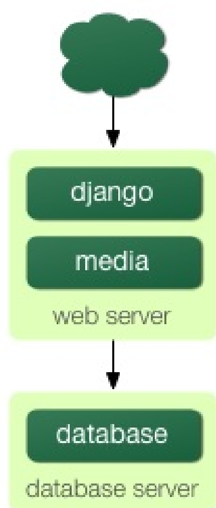


图 20-2： 将数据库移到单独的服务器上。

这里，我们开始步入 *n-tier* 构架。 不要被这个词所吓坏，它只是说明了Web栈的不同部分，被分离到了不同的物理机器上。

我们再来看，如果发现需要不止一台的数据库服务器，考虑使用连接池和数据库备份将是一个好主意。 不幸的是，本书没有足够的时间来讨论这个问题，所以你参考数据库文档或者向社区求助。

运行一个独立的媒体服务器

使用单机服务器仍然留下了一个大问题： 处理动态内容的媒体资源，也是在同一台机器上完成的。

这两个活动是在不同的条件下进行的，因此把它们强行凑和在同一台机器上，你不可能获得很好的性能。 下一步，我们要把媒体资源（任何 不是 由Django视图产生的东西）分离到别的服务器上（请看图20-3）。

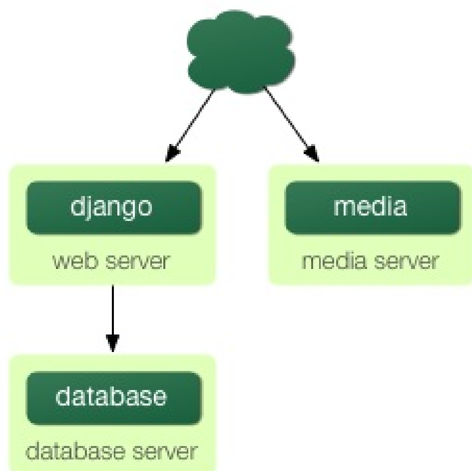


图 20-3： 分离出媒体服务器。

理想的情况是，这个媒体服务器是一个定制的Web服务器，为传送静态媒体资源做了优化。lighttpd和tux (http://www.djangoproject.com/r/tux/) 都是极佳的选择，当然瘦身的Apache服务器也可以工作的很好。

对于拥有大量静态内容（照片、视频等）的站点来说，将媒体服务器分离出去显然有着更加重要的意义，而且应该是扩大规模的时候所要采取的 `第一步措施` 。

这一步需要一点点技巧，Django的admin管理接口需要能够获得足够的权限来处理上传的媒体（通过设置 `MEDIA_ROOT` ）。如果媒体资源在另外的一台服务器上，你需要获得通过网络写操作的权限。 如果你的应用牵涉到文件上载，Django需要能够面向媒体服务器撰写上载媒体 如果媒体是在另外一台服务器上的，你需要部署一种方法使得Django可以通过网络去写这些媒体。

实现负担均衡和数据冗余备份

现在，我们已经尽可能地进行了分解。 这种三台服务器的构架可以承受很大的流量，比如每天1000万的点击率。

这是个好主意。 请看图 20-3，一旦三个服务器中的任何一个发生了故障，你就得关闭整个站点。 因此在引入冗余备份的时候，你并不只是增加了容量，同时也增加了可靠性。

我们首先来考虑Web服务器的点击量。 把同一个Django的站点复制多份，在多台机器上同时运行很容易，我们也只需要同时运行多台机器上的Apache服务器。

你还需要另一个软件来帮助你在多台服务器之间均衡网络流量： 流量均衡器（*load balancer*） 。你可以购买昂贵的专有的硬件均衡器，当然也有一些高质量的开源的软件均衡器可供选择。

Apaches 的 `mod_proxy` 是一个可以考虑的选择，但另一个配置更棒的选择是： memcached是同一个团队的人写的一个负载均衡和反向代理的程序.(见第15章)

记录

如果你使用FastCGI，你同样可以分离前台的web服务器，并在多台其他机器上运行FastCGI服务器来实现相同的负载均衡的功能。 前台的服务器就相当于是一个均衡器，而后台的FastCGI服务进程代替了Apache/mod_python/Django服务器。

现在我们拥有了服务器集群，我们的构架慢慢演化，越来越复杂，如图20-4。

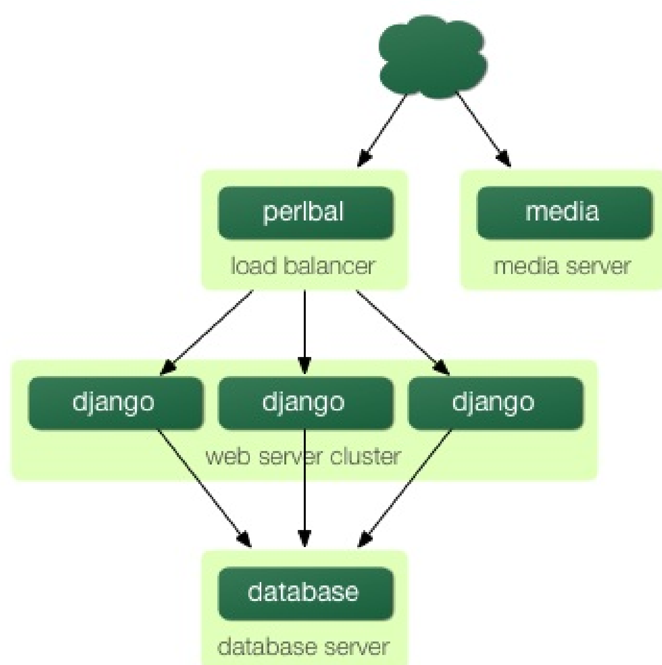


图 20-4： 负载均衡的服务器设置。

值得一提的是，在图中，Web服务器指的是一个集群，来表示许多数量的服务器。 一旦你拥有了一个前台的均衡器，你就可以很方便地增加和删除后台的Web服务器，而且不会造成任何网站不可用的时间。

慢慢变大

下面的这些步骤都是上面最后一个的变体：

- 当你需要更好的数据库性能，你可能需要增加数据库的冗余服务器。 MySQL内置了备份功能；PostgreSQL应该看一下Slony (http://www.djangoproject.com/r/slony/) 和 pgpool (http://www.djangoproject.com/r/pgpool/)，这两个分别是数据库备份和连接池的工具。
- 如果单个均衡器不能达到要求，你可以增加更多的均衡器，并且使用轮训（round-robin）DNS来实现分布访问。
- 如果单台媒体服务器不够用，你可以增加更多的媒体服务器，并通过集群来分布流量。
- 如果你需要更多的高速缓存（cache），你可以增加cache服务器。
- 在任何情况下，只要集群工作性能不好，你都可以往上增加服务器。

重复了几次以后，一个大规模的构架会像图20-5。

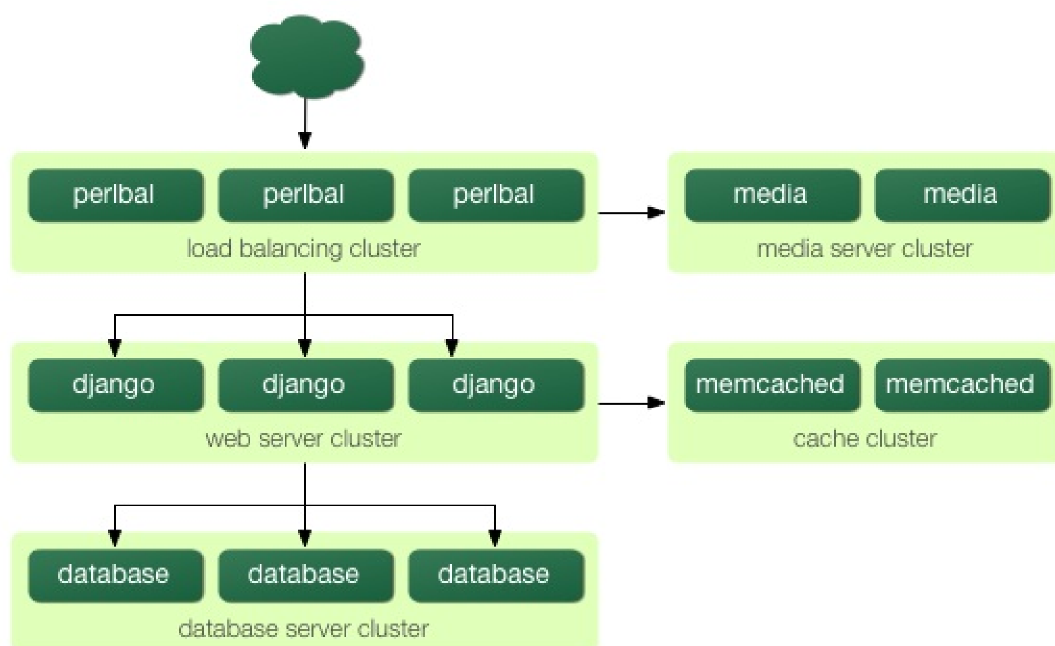


图 20-5。 大规模的Django安装。

尽管我们只是在每一层上展示了两到三台服务器，你可以在上面随意地增加更多。

性能优化

如果你有大笔大笔的钱，遇到扩展性问题时，你可以简单地投资硬件。 对于剩下的人来说，性能优化就是必须要做的一件事。

注意

顺便提一句，谁要是有大笔大笔的钞票，请捐助一点Django项目。 我们也接受未切割的钻石和金币。

不幸的是，性能优化比起科学来说更像是一种艺术，并且这比扩展性更难描述。 如果你真想要构建一个大规模的Django应用，你需要花大量的时间和精力学习如何优化构架中的每一部分。

以下部分总结了多年以来的经验，是一些专属于Django的优化技巧。

RAM怎么也不嫌多

最近即使那些昂贵的RAM也相对来说可以负担的起了。 购买尽可能多的RAM，再在别的上面投资一点点。

高速的处理器并不会大幅度地提高性能；大多数的Web服务器90%的时间都浪费在了硬盘IO上。 当硬盘上的数据开始交换，性能就急剧下降。 更快速的硬盘可以改善这个问题，但是比起RAM来说，那太贵了。

如果你拥有多台服务器，首要的是要在数据库服务器上增加内存。 如果你能负担得起，把你整个数据库都放入到内存中。 这应该不是很困难，我们已经开发过一个站点上面有多于一百万条报刊文章，这个站点使用了不到2GB的空间。

下一步，最大化Web服务器上的内存。 最理想的情况是，没有一台服务器进行磁盘交换。 如果你达到了这个水平，你就能应付大多数正常的流量。

禁用 Keep-Alive

`Keep-Alive` 是HTTP提供的功能之一，它的目的是允许多个HTTP请求复用一个TCP连接，也就是允许在同一个TCP连接上发起多个HTTP请求，这样有效的避免了每个HTTP请求都重新建立自己的TCP连接的开销。

这一眼看上去是好事，但它足以杀死Django站点的性能。 如果你从单独的媒体服务器上向用户提供服务，每个光顾你站点的用户都大约10秒钟左右发出一次请求。 这就使得HTTP服务器一直在等待下一次keep-alive 的请求，空闲的HTTP服务器和工作时消耗一样多的内存。

使用 memcached

尽管Django支持多种不同的cache后台机制，没有一种的性能可以 接近 memcached。 如果你有一个高流量的站点，不要犹豫，直接选择memcached。

经常使用memcached

当然，选择了memcached而不去使用它，你不会从中获得任何性能上的提升。 Chapter 15 is your best friend here: 学习如何使用Django的cache框架，并且尽可能地使用它。 大量的可抢占式的高速缓存通常是一个站点在大流量下正常工作的唯一瓶颈。

参加讨论

Django相关的每一个部分，从Linux到Apache到PostgreSQL或者MySQL背后，都有一个非常棒的社区支持。 如果你真想从你的服务器上榨干最后1%的性能，加入开源社区寻求帮助。 多数的自由软件社区成员都会很乐意地提供帮助。

别忘了Django社区。 这本书谦逊的作者只是Django开发团队中的两位成员。 我们的社区有大量的经验可以提供。

下一章

下面的章节集中在其他的一些Django特性上，你是否需要它们取决于你的应用项目。 可以自由选择阅读。

第十三章：输出非HTML内容

通常当我们谈到开发网站时，主要谈论的是HTML。当然，Web远不只有HTML，我们在Web上用多种格式来发布数据：RSS、PDF、图片等。

到目前为止，我们的注意力都是放在常见 HTML 代码生成上，但是在这一章中，我们将会对使用 Django 生成其它格式的内容进行简要介绍。

Django拥有一些便利的内建工具帮助你生成常见的非HTML内容：

- RSS/Atom 聚合文件
- 站点地图 （一个XML格式文件，最初由Google开发，用于给搜索引擎提示线索）

我们稍后会逐一研究这些工具，不过首先让我们来了解些基础原理。

基础：视图和MIME类型

回顾一下第三章，视图函数只是一个以Web请求为参数并返回Web响应的Python函数。这个响应可以是一个Web页面的HTML内容，或者一个跳转，或者一个404 错误，或者一个XML文档，或者一幅图片，或者映射到任何东西上。

更正式的说，一个Django视图函数 必须

- 接受一个 `HttpRequest` 实例作为它的第一个参数
- 返回一个 `HttpResponse` 实例

从一个视图返回一个非 HTML 内容的关键是在构造一个 `HttpResponse` 类时，需要指定 `mimetype` 参数。通过改变 MIME 类型，我们可以通知浏览器将要返回的数据是另一种类型。

下面我们以返回一张PNG图片的视图为例。为了使事情能尽可能的简单，我们只是读入一张存储在磁盘上的图片：

```
from django.http import HttpResponse

def my_image(request):
    image_data = open("/path/to/my/image.png", "rb").read()
    return HttpResponse(image_data, mimetype="image/png")
```

就是这么简单。如果改变 `open()` 中的图片路径为一张真实图片的路径，那么就可以使用这个十分简单的视图来提供一张图片，并且浏览器可以正确显示它。

另外我们必须了解的是 `HttpResponse` 对象实现了Python标准的文件应用程序接口(API)。 这就是说你可以在Python（或第三方库）任何用到文件的地方使用”HttpResponse”实例。

下面将用 Django 生成 CSV 文件为例，说明它的工作原理。

生成 **CSV** 文件

CSV 是一种简单的数据格式，通常为电子表格软件所使用。 它主要是由一系列的表格行组成，每行中单元格之间使用逗号(CSV 是 逗号分隔数值*(comma-separated values)* 的缩写)隔开。例如，下面是CSV格式的“不守规矩”的飞机乘客表。

```
Year,Unruly Airline Passengers
1995,146
1996,184
1997,235
1998,200
1999,226
2000,251
2001,299
2002,273
2003,281
2004,304
2005,203
2006,134
2007,147
```

备注

前面的列表包含真实数据。 这些数据来自美国 联邦航空管理局。

CSV格式尽管看起来简单，却是全球通用的。 但是不同的软件会生成和使用不同的 CSV 的变种，在使用上会有一些不便。 幸运的是， Python 使用的是标准 CSV 库， `csv` ，所以它更通用。

因为 `csv` 模块操作的是类似文件的对象，所以可以使用 `HttpResponse` 替换：

```
import csv
from django.http import HttpResponse

# Number of unruly passengers each year 1995 - 2005\. In a real application
# this would likely come from a database or some other back-end data store.
UNRULY_PASSENGERS = [146,184,235,200,226,251,299,273,281,304,203]

def unruly_passengers_csv(request):
    # Create the HttpResponse object with the appropriate CSV header.
    response = HttpResponse(mimetype='text/csv')
    response['Content-Disposition'] = 'attachment; filename=unruly.csv'

    # Create the CSV writer using the HttpResponse as the "file."
    writer = csv.writer(response)
    writer.writerow(['Year', 'Unruly Airline Passengers'])
    for (year, num) in zip(range(1995, 2006), UNRULY_PASSENGERS):
        writer.writerow([year, num])

    return response
```

代码和注释可以说是很清楚，但还有一些事情需要特别注意：

> 响应返回的是 `text/csv` MIME类型（而非默认的 `text/html` ）。这会告诉浏览器，返回的文档是CSV文件。
>
> 响应会有一个附加的 `Content-Disposition` 头部，它包含有CSV文件的文件名。 这个头部（或者说，附加部分）会指示浏览器弹出对话框询问文件存放的位置（而不仅仅是显示）。 这个文件名是任意的。 它会显示在浏览器的另存为对话框中。
>
> 要在 `HttpResponse` 指定头部信息，只需把 `HttpResponse` 当做字典使用就可以了。
>
> 与创建CSV的应用程序界面（API）挂接是很容易的： 只需将 `response` 作为第一个变量传递给 `csv.writer` 。 `csv.writer` 函数需要一个文件类的对象， `HttpResponse` 正好能达成这个目的。
>
> 调用 `writer.writerow` ，并且传递给它一个类似 list 或者 tuple 的可迭代对象，就可以在 CSV 文件中写入一行。
>
> CSV 模块考虑到了引用的问题，所以您不用担心逸出字符串中引号和逗号。 只要把信息传递给 `writerow()` ，它会处理好所有的事情。

在任何需要返回非 HTML 内容的时候，都需要经过以下几步： 创建一个 `HttpResponse` 响应对象（需要指定特殊的 MIME 类型），它它传给需要处理文件的函数，然后返回这个响应对象。

下面是一些其它的例子。

生成 PDF 文件

便携文档格式 (PDF) 是由 Adobe 开发的格式，主要用于呈现可打印的文档，其中包含有pixel-perfect 格式，嵌入字体以及2D矢量图像。 You can think of a PDF document as the digital equivalent of a printed document; indeed, PDFs are often used in distributing documents for the purpose of printing them.

可以方便的使用 Python 和 Django 生成 PDF 文档需要归功于一个出色的开源库， ReportLab (http://www.reportlab.org/rl_toolkit.html) 。动态生成 PDF 文件的好处是在不同的情况下，如不同的用户或者不同的内容，可以按需生成不同的 PDF 文件。 The advantage of generating PDF files dynamically is that you can create customized PDFs for different purposes say, for different users or different pieces of content.

下面的例子是使用 Django 和 ReportLab 在 KUSports.com 上生成个性化的可打印的 NCAA 赛程表 (tournament brackets) 。

安装 ReportLab

在生成 PDF 文件之前，需要安装 ReportLab 库。这通常是个很简单的过程： Its usually simple: just download and install the library from http://www.reportlab.org/downloads.html.

Note

如果使用的是一些新的 Linux 发行版，则在安装前可以先检查包管理软件。 多数软件包仓库中都加入了 ReportLab 。

比如，如果使用（杰出的） Ubuntu 发行版，只需要简单的 `apt-get install python-reportlab` 一行命令即可完成安装。

使用手册（原始的只有 PDF 格式）可以从 >http://www.reportlab.org/rsrc/userguide.pdf 下载，其中包含有一些其它的安装指南。

在 Python 交互环境中导入这个软件包以检查安装是否成功。

```
>>> import reportlab
```

如果刚才那条命令没有出现任何错误，则表明安装成功。

编写视图

和 CSV 类似，由 Django 动态生成 PDF 文件很简单，因为 ReportLab API 同样可以使用类似文件对象。

下面是一个 Hello World 的示例：

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def hello_pdf(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=hello.pdf'

    # Create the PDF object, using the response object as its "file."
    p = canvas.Canvas(response)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly, and we're done.
    p.showPage()
    p.save()
    return response
```

需要注意以下几点：

- 这里我们使用的 MIME 类型是 `application/pdf` 。这会告诉浏览器这个文档是一个 PDF 文档，而不是 HTML 文档。 如果忽略了这个参数，浏览器可能会把这个文件看成 HTML 文档，这会使浏览器的窗口中出现很奇怪的文字。 If you leave off this information,

browsers will probably interpret the response as HTML, which will result in scary gobbledygook in the browser window.

- 使用 ReportLab 的 API 很简单：只需要将 `response` 对象作为 `canvas.Canvas` 的第一个参数传入。
- 所有后续的 PDF 生成方法需要由 PDF 对象调用（在本例中是 `p` ），而不是 `response` 对象。
- 最后需要对 PDF 文件调用 `showPage()` 和 `save()` 方法（否则你会得到一个损坏的 PDF 文件）。

复杂的 PDF 文件

如果您在创建一个复杂的 PDF 文档（或者任何较大的数据块），请使用 `cStringIO` 库存放临时生成的 PDF 文件。 `cStringIO` 提供了一个用 C 编写的类似文件对象的接口，从而可以使系统的效率最高。

下面是使用 `cStringIO` 重写的 Hello World 例子：

```
from cStringIO import StringIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def hello_pdf(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=hello.pdf'

    temp = StringIO()

    # Create the PDF object, using the StringIO object as its "file."
    p = canvas.Canvas(temp)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly.
    p.showPage()
    p.save()

    # Get the value of the StringIO buffer and write it to the response.
    response.write(temp.getvalue())
    return response
```

其它的可能性

使用 Python 可以生成许多其它类型的内容，下面介绍的是一些其它的想法和一些可以用以实现它们的库。 Here are a few more ideas and some pointers to libraries you could use to implement them:

> *ZIP* 文件 ：Python 标准库中包含有 `zipfile` 模块，它可以读和写压缩的 ZIP 文件。 它可以用于按需生成一些文件的压缩包，或者在需要时压缩大的文档。 如果是 TAR 文件则可以使用标准库 `tarfile` 模块。
>
> 动态图片 ： Python 图片处理库 (PIL; http://www.pythonware.com/products/pil/) 是极好的生成图片(PNG, JPEG, GIF 以及其它许多格式)的工具。 它可以用于自动为图片生成缩略图，将多张图片压缩到单独的框架中，或者是做基于 Web 的图片处理。
>
> 图表 ： Python 有许多出色并且强大的图表库用以绘制图表，按需地图，表格等。 我们不可能将它们全部列出，所以下面列出的是个中的翘楚。
>
> - `matplotlib` (http://matplotlib.sourceforge.net/) 可以用于生成通常是由 matlab 或者 Mathematica 生成的高质量图表。
> - `pygraphviz` (https://networkx.lanl.gov/wiki/pygraphviz) 是一个 Graphviz 图形布局的工具 (http://graphviz.org/) 的 Python 接口，可以用于生成结构化的图表和网络。

总之，所有可以写文件的库都可以与 Django 同时使用。 The possibilities are immense.

我们已经了解了生成“非HTML”内容的基本知识，让我们进一步总结一下。 Django拥有很多用以生成各类“非HTML”内容的内置工具。

内容聚合器应用框架

Django带来了一个高级的聚合生成框架，它使得创建RSS和Atom feeds变得非常容易。

什么是RSS？ 什么是Atom？

RSS和Atom都是基于XML的格式，你可以用它来提供有关你站点内容的自动更新的feed。 了解更多关于RSS的可以访问 http://www.whatisrss.com/, 更多Atom的信息可以访问 http://www.atomenabled.org/.

想创建一个联合供稿的源(syndication feed)，所需要做的只是写一个简短的python类。 你可以创建任意多的源(feed)。

高级feed生成框架是一个默认绑定到/feeds/的视图，Django使用URL的其它部分(在/feeds/之后的任何东西)来决定输出 哪个feed Django uses the remainder of the URL (everything after `/feeds/`) to determine which feed to return.

要创建一个 sitemap，你只需要写一个 `Sitemap` 类然后配置你的URLconf指向它。

初始化

为了在您的Django站点中激活syndication feeds, 添加如下的 URLconf:

```
(r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
    {'feed_dict': feeds}
),
```

这一行告诉Django使用RSS框架处理所有的以 `"feeds/"` 开头的URL. (你可以修改 `"feeds/"` 前缀以满足您自己的要求.)

URLConf里有一行参数： `{'feed_dict': feeds}` ，这个参数可以把对应URL需要发布的feed内容传递给 syndication framework

特别的，feed_dict应该是一个映射feed的slug(简短URL标签)到它的Feed类的字典 你可以在URL配置本身里定义feed_dict，这里是一个完整的例子 You can define the `feed_dict` in the URLconf itself. Here's a full example URLconf:

```
from django.conf.urls.defaults import *
from mysite.feeds import LatestEntries, LatestEntriesByCategory

feeds = {
    'latest': LatestEntries,
    'categories': LatestEntriesByCategory,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

前面的例子注册了两个feed:

- `LatestEntries``表示的内容将对应到``feeds/latest/` .
- `LatestEntriesByCategory``的内容将对应到 ``feeds/categories/` .

以上的设定完成之后，接下来需要自己定义 `Feed` 类

一个 `Feed` 类是一个简单的python类，用来表示一个syndication feed. 一个feed可能是简单的 (例如一个站点新闻feed，或者最基本的，显示一个blog的最新条目)，也可能更加复杂(例如一个显示blog某一类别下所有条目的feed。 这里类别 category 是个变量).

Feed类必须继承django.contrib.syndication.feeds.Feed，它们可以在你的代码树的任何位置

一个简单的Feed

This simple example describes a feed of the latest five blog entries for a given blog:

```
from django.contrib.syndication.feeds import Feed
from mysite.blog.models import Entry

class LatestEntries(Feed):
    title = "My Blog"
    link = "/archive/"
    description = "The latest news about stuff."

    def items(self):
        return Entry.objects.order_by('-pub_date')[:5]
```

要注意的重要的事情如下所示:

- 子类 `django.contrib.syndication.feeds.Feed` .
- `title` , `link` , 和 `description` 对应一个标准 RSS 里的 `<title>` , `<link>` , 和 `<description>` 标签.
- `items()` 是一个方法，返回一个用以包含在包含在feed的 `<item>` 元素里的 list 虽然例子里用Djangos database API返回的 `NewsItem` 对象, `items()` 不一定必须返回model的实例 Although this example returns `Entry` objects using Django's database API, `items()` doesn't have to return model instances.

还有一个步骤，在一个RSS feed里，每个(item)有一个(title)，(link)和(description)，我们需要告诉框架 把数据放到这些元素中 In an RSS feed, each `<item>` has a `<title>` , `<link>` , and `<description>` . We need to tell the framework what data to put into those elements.

> 如果要指定 `<title>` 和 `<description>` ，可以建立一个Django模板（见Chapter 4）名字叫 `feeds/latest_title.html` 和 `feeds/latest_description.html` ，后者是URLConf里为对应feed指定的 `slug` 。注意 `.html` 后缀是必须的。 Note that the `.html` extension is required.
>
> RSS系统模板渲染每一个条目，需要给传递2个参数给模板上下文变量：
>
> - `obj` ：当前对象 (返回到 `items()` 任意对象之一)。
> - `site` ：一个表示当前站点的 `django.models.core.sites.Site` 对象。 这对于 `{{ site.domain }}` 或者 `{{ site.name }}` 很有用。
>
> 如果你在创建模板的时候，没有指明标题或者描述信息，框架会默认使用 `"{{ obj }}"` ，对象的字符串表示。 (For model objects, this will be the `__unicode__()` method.
>
> 你也可以通过修改 `Feed` 类中的两个属性 `title_template` 和 `description_template` 来改变这两个模板的名字。
>
> 你有两种方法来指定 `<link>` 的内容。 Django 首先执行 `items()` 中每一项的 `get_absolute_url()` 方法。 如果该方法不存在，就会尝试执行 `Feed` 类中的 `item_link()` 方法，并将自身作为 `item` 参数传递进去。
>
> `get_absolute_url()` 和 `item_link()` 都应该以Python字符串形式返回URL。
>
> 对于前面提到的 `LatestEntries` 例子，我们可以实现一个简单的feed模板。 `latest_title.html` 包括：

```
{{ obj.title }}
```

> 并且 `latest_description.html` 包含:

```
{{ obj.description }}
```

> 这真是 太 简单了！

一个更复杂的Feed

框架通过参数支持更加复杂的feeds。

For example, say your blog offers an RSS feed for every distinct tag you've used to categorize your entries. 如果为每一个单独的区域建立一个 `Feed` 类就显得很不明智。

取而代之的方法是，使用聚合框架来产生一个通用的源，使其可以根据feeds URL返回相应的信息。

Your tag-specific feeds could use URLs like this:

- `http://example.com/feeds/tags/python/` : Returns recent entries tagged with python
- `http://example.com/feeds/tags/cats/` : Returns recent entries tagged with cats

固定的那一部分是 `"beats"` （区域）。

举个例子会澄清一切。 下面是每个地区特定的feeds：

```
from django.core.exceptions import ObjectDoesNotExist
from mysite.blog.models import Entry, Tag

class TagFeed(Feed):
    def get_object(self, bits):
        # In case of "/feeds/tags/cats/dogs/mice/", or other such
        # clutter, check that bits has only one member.
        if len(bits) != 1:
            raise ObjectDoesNotExist
        return Tag.objects.get(tag=bits[0])

    def title(self, obj):
        return "My Blog: Entries tagged with %s" % obj.tag

    def link(self, obj):
        return obj.get_absolute_url()

    def description(self, obj):
        return "Entries tagged with %s" % obj.tag

    def items(self, obj):
        entries = Entry.objects.filter(tags__id__exact=obj.id)
        return entries.order_by('-pub_date')[:30]
```

以下是RSS框架的基本算法，我们假设通过URL `/rss/beats/0613/` 来访问这个类：

> 框架获得了URL `/rss/beats/0613/` 并且注意到URL中的slug部分后面含有更多的信息。它将斜杠(`"/"`)作为分隔符，把剩余的字符串分割开作为参数，调用 `Feed` 类的 `get_object()` 方法。
>
> 在这个例子中，添加的信息是 `['0613']` 。对于 `/rss/beats/0613/foo/bar/` 的一个URL请求， 这些信息就是 `['0613', 'foo', 'bar']` 。
>
> `get_object()` 就根据给定的 `bits` 值来返回区域信息。
>
> In this case, it uses the Django database API to retrieve the `Tag` . Note that `get_object()` should raise `django.core.exceptions.ObjectDoesNotExist` if given invalid parameters. 在 `Beat.objects.get()` 调用中也没有出现 `try` / `except` 代码块。 函数在出错时抛出 `Beat.DoesNotExist` 异常，而 `Beat.DoesNotExist` 是 `ObjectDoesNotExist` 异常的一个子类型。
>
> 为产生 <title> ， <link> ， 和 <description> 的feeds， Django使用 `title()` , `link()` , 和 `description()` 方法。 在上面的例子中，它们都是简单的字符串类型的类属性，而这个例子表明，它们既可以是字符串， 也可以是 方法。 对于每一个 `title` ， `link` 和 `description` 的组合，Django使用以下的算法：
>
> 1. 试图调用一个函数，并且以 `get_object()` 返回的对象作为参数传递给 `obj` 参数。
>
> 1. 如果没有成功，则不带参数调用一个方法。
>
> 1. 还不成功，则使用类属性。
>
> 最后，值得注意的是，这个例子中的 `items()` 使用 `obj` 参数。 对于 `items` 的算法就如同上面第一步所描述的那样，首先尝试 `items(obj)` ， 然后是 `items()` ，最后是 `items` 类属性（必须是一个列表）。

`Feed` 类所有方法和属性的完整文档，请参考官方的Django文档 (http://www.djangoproject.com/documentation/0.96/syndication_feeds/) 。

指定Feed的类型

默认情况下, 聚合框架生成RSS 2.0. 要改变这样的情况, 在 `Feed` 类中添加一个 `feed_type` 属性. To change that, add a `feed_type` attribute to your `Feed` class:

```
from django.utils.feedgenerator import Atom1Feed

class MyFeed(Feed):
    feed_type = Atom1Feed
```

注意你把 `feed_type` 赋值成一个类对象，而不是类实例。 目前合法的Feed类型如表11-1所示。

表 11-1\. Feed 类型

Feed 类	类型
`django.utils.feedgenerator.Rss201rev2Feed`	RSS 2.01 (default)
`django.utils.feedgenerator.RssUserland091Feed`	RSS 0.91
`django.utils.feedgenerator.Atom1Feed`	Atom 1.0

闭包

为了指定闭包（例如，与feed项比方说MP3 feeds相关联的媒体资源信息），使用 item_enclosure_url ， item_enclosure_length ， 以及 item_enclosure_mime_type ，比如

```
from myproject.models import Song

class MyFeedWithEnclosures(Feed):
    title = "Example feed with enclosures"
    link = "/feeds/example-with-enclosures/"

    def items(self):
        return Song.objects.all()[:30]

    def item_enclosure_url(self, item):
        return item.song_url

    def item_enclosure_length(self, item):
        return item.song_length

    item_enclosure_mime_type = "audio/mpeg"
```

当然，你首先要创建一个包含有 song_url 和 song_length （比如按照字节计算的长度）域的 Song 对象。

语言

聚合框架自动创建的Feed包含适当的 <language> 标签(RSS 2.0) 或 xml:lang 属性(Atom). 他直接来自于您的 LANGUAGE_CODE 设置. This comes directly from your LANGUAGE_CODE setting.

URLs

link 方法/属性可以以绝对URL的形式（例如， "/blog/" ）或者指定协议和域名的URL的形式返回（例如 "http://www.example.com/blog/" ）。如果 link 没有返回域名，聚合框架会根据 SITE_ID 设置，自动的插入当前站点的域信息。 (See Chapter 16 for more on SITE_ID and the sites framework.)

Atom feeds需要 <link rel="self"> 指明feeds现在的位置。 The syndication framework populates this automatically.

同时发布Atom and RSS

一些开发人员想 同时 支持Atom和RSS。 这在Django中很容易实现： 只需创建一个你的 `feed` 类的子类，然后修改 `feed_type` ，并且更新URLconf内容。 下面是一个完整的例子： Here's a full example:

```
from django.contrib.syndication.feeds import Feed
from django.utils.feedgenerator import Atom1Feed
from mysite.blog.models import Entry

class RssLatestEntries(Feed):
    title = "My Blog"
    link = "/archive/"
    description = "The latest news about stuff."

    def items(self):
        return Entry.objects.order_by('-pub_date')[:5]

class AtomLatestEntries(RssLatestEntries):
    feed_type = Atom1Feed
```

这是与之相对应那个的URLconf：

```
from django.conf.urls.defaults import *
from myproject.feeds import RssLatestEntries, AtomLatestEntries

feeds = {
    'rss': RssLatestEntries,
    'atom': AtomLatestEntries,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

Sitemap 框架

sitemap 是你服务器上的一个XML文件，它告诉搜索引擎你的页面的更新频率和某些页面相对于其它页面的重要性。 这个信息会帮助搜索引擎索引你的网站。

例如，这是 Django 网站(http://www.djangoproject.com/sitemap.xml)sitemap的一部分：

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.djangoproject.com/documentation/</loc>
    <changefreq>weekly</changefreq>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>http://www.djangoproject.com/documentation/0_90/</loc>
    <changefreq>never</changefreq>
    <priority>0.1</priority>
  </url>
  ...
</urlset>
```

需要了解更多有关 sitemaps 的信息, 请参见 http://www.sitemaps.org/.

Django sitemap 框架允许你用 Python 代码来表述这些信息，从而自动创建这个XML文件。要创建一个站点地图，你只需要写一个 `Sitemap` 类，并且在URLconf中指向它。

安装

要安装 sitemap 应用程序, 按下面的步骤进行:

1. 将 `'django.contrib.sitemaps'` 添加到您的 `INSTALLED_APPS` 设置中.
2. 确保 `'django.template.loaders.app_directories.load_template_source'` 在您的 `TEMPLATE_LOADERS` 设置中。 默认情况下它在那里, 所以, 如果你已经改变了那个设置的话, 只需要改回来即可。
3. 确定您已经安装了 sites 框架 (参见第14章).

Note

sitemap 应用程序没有安装任何数据库表. 它需要加入到 `INSTALLED_APPS` 中的唯一原因是: 这样 `load_template_source` 模板加载器可以找到默认的模板. The only reason it needs to go into `INSTALLED_APPS` is so the `load_template_source` template loader can find the default templates.

Initialization

要在您的Django站点中激活sitemap生成, 请在您的 URLconf 中添加这一行:

```
(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps})
```

This line tells Django to build a sitemap when a client accesses `/sitemap.xml` . Note that the dot character in `sitemap.xml` is escaped with a backslash, because dots have a special meaning in regular expressions.

sitemap文件的名字无关紧要，但是它在服务器上的位置却很重要。 搜索引擎只索引你的sitemap中当前URL级别及其以下级别的链接。 用一个实例来说，如果 `sitemap.xml` 位于你的根目录，那么它将引用任何的URL。 然而，如果你的sitemap位于 `/content/sitemap.xml` ，那么它只引用以 `/content/` 打头的URL。

sitemap视图需要一个额外的必须的参数： `{'sitemaps': sitemaps}` . `sitemaps` should be a dictionary that maps a short section label (e.g., `blog` or `news`) to its `Sitemap` class (e.g., `BlogSitemap` or `NewsSitemap`). It may also map to an *instance* of a `Sitemap` class (e.g., `BlogSitemap(some_var)`).

Sitemap 类

`Sitemap` 类展示了一个进入地图站点简单的Python类片断.例如,一个 `Sitemap` 类能展现所有日志入口，而另外一个能够调度所有的日历事件。 For example, one `Sitemap` class could represent all the entries of your weblog, while another could represent all of the events in your events calendar.

在最简单的例子中，所有部分可以全部包含在一个 `sitemap.xml` 中，也可以使用框架来产生一个站点地图，为每一个独立的部分产生一个单独的站点文件。

`Sitemap` 类必须是 `django.contrib.sitemaps.Sitemap` 的子类. 他们可以存在于您的代码树的任何地方。

例如假设你有一个blog系统，有一个 `Entry` 的model，并且你希望你的站点地图包含所有连到你的blog入口的超链接。 你的 `Sitemap` 类很可能是这样的：

```
from django.contrib.sitemaps import Sitemap
from mysite.blog.models import Entry

class BlogSitemap(Sitemap):
    changefreq = "never"
    priority = 0.5

    def items(self):
        return Entry.objects.filter(is_draft=False)

    def lastmod(self, obj):
        return obj.pub_date
```

声明一个 `Sitemap` 和声明一个 `Feed` 看起来很类似；这都是预先设计好的。

如同 `Feed` 类一样， `Sitemap` 成员也既可以是方法，也可以是属性。 想要知道更详细的内容，请参见上文 《一个复杂的例子》章节。

一个 `Sitemap` 类可以定义如下 方法/属性:

> `items` (必需)：提供对象列表。 框架并不关心对象的 类型 ；唯一关心的是这些对象会传递给 `location()` ， `lastmod()` ， `changefreq()` ，和 `priority()` 方法。
>
> `location` （可选）： 给定对象的绝对URL。 绝对URL不包含协议名称和域名。 下面是一些例子：
>
> - 好的: `'/foo/bar/'` `'/foo/bar/'`
> - 差的: `'example.com/foo/bar/'` `'example.com/foo/bar/'`
> - Bad: `'http://example.com/foo/bar/'`
>
> 如果没有提供 `location` , 框架将会在每个 `items()` 返回的对象上调用 `get_absolute_url()` 方法.
>
> `lastmod` (可选): 对象的最后修改日期, 作为一个Python `datetime` 对象. The object's last modification date, as a Python `datetime` object.
>
> `changefreq` （可选）： 对象变更的频率。 可选的值如下（详见Sitemaps文档）：
>
> - `'always'`
> - `'hourly'`
> - `'daily'`
> - `'weekly'`
> - `'monthly'`
> - `'yearly'`
> - `'never'`
>
> `priority` （可选）： 取值范围在 `0.0` and `1.0` 之间，用来表明优先级。

快捷方式

sitemap框架提供了一些常用的类。 在下一部分中会看到。

FlatPageSitemap

`django.contrib.sitemaps.FlatPageSitemap` 类涉及到站点中所有的flat page，并在sitemap中建立一个入口。 但仅仅只包含 `location` 属性，不支持 `lastmod` ， `changefreq` ，或者 `priority` 。

参见第16章获取有关flat page的更多的内容.

GenericSitemap

`GenericSitemap` 与所有的通用视图一同工作（详见第9章）。

你可以如下使用它，创建一个实例，并通过 `info_dict` 传递给通用视图。 唯一的要求是字典包含 `queryset` 这一项。 也可以用 `date_field` 来指明从 `queryset` 中取回的对象的日期域。 这会被用作站点地图中的 `lastmod` 属性。

下面是一个使用 `FlatPageSitemap` and `GenericSiteMap` （包括前面所假定的 `Entry` 对象）的URLconf：

```
from django.conf.urls.defaults import *
from django.contrib.sitemaps import FlatPageSitemap, GenericSitemap
from mysite.blog.models import Entry

info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}

sitemaps = {
    'flatpages': FlatPageSitemap,
    'blog': GenericSitemap(info_dict, priority=0.6),
}

urlpatterns = patterns('',
    # some generic view using info_dict
    # ...

    # the sitemap
    (r'^sitemap\.xml$',
     'django.contrib.sitemaps.views.sitemap',
     {'sitemaps': sitemaps})
)
```

创建一个Sitemap索引

sitemap框架同样可以根据 `sitemaps` 字典中定义的单独的sitemap文件来建立索引。 用法区别如下：

- 您在您的URLconf 中使用了两个视图: `django.contrib.sitemaps.views.index` 和 `django.contrib.sitemaps.views.sitemap` . `django.contrib.sitemaps.views.index` 和 `django.contrib.sitemaps.views.sitemap`
- `django.contrib.sitemaps.views.sitemap` 视图需要带一个 `section` 关键字参数.

这里是前面的例子的相关的 URLconf 行看起来的样子:

```
(r'^sitemap.xml$',
 'django.contrib.sitemaps.views.index',
 {'sitemaps': sitemaps}),

(r'^sitemap-(?P<section>.+).xml$',
 'django.contrib.sitemaps.views.sitemap',
 {'sitemaps': sitemaps})
```

这将自动生成一个 `sitemap.xml` 文件, 它同时引用 `sitemap-flatpages.xml` 和 `sitemap-blog.xml` . `Sitemap` 类和 `sitemaps` 目录根本没有更改.

通知Google

当你的sitemap变化的时候，你会想通知Google，以便让它知道对你的站点进行重新索引。框架就提供了这样的一个函数： `django.contrib.sitemaps.ping_google()` 。

`ping_google()` 有一个可选的参数 `sitemap_url` ，它应该是你的站点地图的URL绝对地址（例如：

如果不能够确定你的sitemap URL, `ping_google()` 会引发 `django.contrib.sitemaps.SitemapNotFound` 异常。

我们可以通过模型中的 `save()` 方法来调用 `ping_google()` ：

```
from django.contrib.sitemaps import ping_google

class Entry(models.Model):
    # ...
    def save(self, *args, **kwargs):
        super(Entry, self).save(*args, **kwargs)
        try:
            ping_google()
        except Exception:
            # Bare 'except' because we could get a variety
            # of HTTP-related exceptions.
            pass
```

一个更有效的解决方案是用 `cron` 脚本或任务调度表来调用 `ping_google()` ，该方法使用Http直接请求Google服务器，从而减少每次调用 `save()` 时占用的网络带宽。 The function makes an HTTP request to Google's servers, so you may not want to introduce that network overhead each time you call `save()` .

Finally, if `'django.contrib.sitemaps'` is in your `INSTALLED_APPS` , then your `manage.py` will include a new command, `ping_google` . This is useful for command-line access to pinging. For example:

```
python manage.py ping_google /sitemap.xml
```

下一章

下面, 我们要继续深入挖掘所有的Django给你的很好的内置工具。 第十四章，查看创建用户自定义站点需要的工具 sessions, users 和authentication.

第十四章： 会话、用户和注册

是时候承认了： 我们有意的避开了Web开发中极其重要的方面。 到目前为止，我们都在假定，网站流量是大量的匿名用户带来的。

这当然不对。 浏览器的背后都是活生生的人(至少某些时候是)。 这忽略了重要的一点： 互联网服务于人而不是机器。 要开发一个真正令人心动的网站，我们必须面对浏览器后面活生生的人。

很不幸，这并不容易。 HTTP被设计为"无状态"，每次请求都处于相同的空间中。 在一次请求和下一次请求之间没有任何状态保持，我们无法根据请求的任何方面(IP地址，用户代理等)来识别来自同一人的连续请求。

在本章中你将学会如何搞定状态的问题。 好了，我们会从较低的层次(*cookies*)开始，然后过渡到用高层的工具来搞定会话，用户和注册的问题。

Cookies

浏览器的开发者在很早的时候就已经意识到， HTTP's 的无状态会对Web开发者带来很大的问题，于是(*cookies*)应运而生。 cookies 是浏览器为 Web 服务器存储的一小段信息。 每次浏览器从某个服务器请求页面时，它向服务器回送之前收到的cookies

来看看它是怎么工作的。 当你打开浏览器并访问 `google.com` ，你的浏览器会给Google发送一个HTTP请求，起始部分就象这样：

```
GET / HTTP/1.1
Host: google.com
...
```

当 Google响应时，HTTP的响应是这样的：

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671;
            expires=Sun, 17-Jan-2038 19:14:07 GMT;
            path=/; domain=.google.com
Server: GWS/2.1
...
```

注意 `Set-Cookie` 的头部。 你的浏览器会存储cookie值(`PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671`)，而且每次访问google 站点都会回送这个cookie值。 因此当你下次访问Google时，你的浏览器会发送像这样的请求：

```
GET / HTTP/1.1
Host: google.com
Cookie: PREF=ID=5b14f22bdaf1e81c:TM=1167000671:LM=1167000671
...
```

于是 `Cookies` 的值会告诉Google，你就是早些时候访问过Google网站的人。 这个值可能是数据库中存储用户信息的key，可以用它在页面上显示你的用户名。 Google会（以及目前）使用它在网页上显示你账号的用户名。

存取Cookies

在Django中处理持久化，大部分时候你会更愿意用高层些的session 和/或 后面要讨论的user框架。 但在此之前，我们需要停下来在底层看看如何读写cookies。 这会帮助你理解本章节后面要讨论的工具是如何工作的，而且如果你需要自己操作cookies，这也会有所帮助。

读取已经设置好的cookies极其简单。 每一个 `HttpRequest` 对象都有一个 `COOKIES` 对象，该对象的行为类似一个字典，你可以使用它读取任何浏览器发送给视图（view）的cookies。

```
def show_color(request):
    if "favorite_color" in request.COOKIES:
        return HttpResponse("Your favorite color is %s" %             request.COOKIES["fa
    else:
        return HttpResponse("You don't have a favorite color.")
```

写cookies稍微复杂点。 你需要使用 `HttpResponse` 对象的 `set_cookie()` 方法。 这儿有个基于 `GET` 参数来设置 `favorite_color`

cookie的例子：

```
def set_color(request):
    if "favorite_color" in request.GET:

        # Create an HttpResponse object...
        response = HttpResponse("Your favorite color is now %s" %             request.GET

        # ... and set a cookie on the response
        response.set_cookie("favorite_color",
                            request.GET["favorite_color"])

        return response

    else:
        return HttpResponse("You didn't give a favorite color.")
```

你可以给 `response.set_cookie()` 传递一些可选的参数来控制cookie的行为，详见表14-1。

System Message: ERROR/3 (`<string>` , line 145)

Error parsing content block for the “table” directive: exactly one table expected.

```
.. table:: 表 14-1: Cookie 选项

   +--------------------------------+-------------------------+----------------------
   |参数                            |缺省值                   |描述
   +================================+=========================+======================
   |``max_age``                     |``None``                 |cookie需要延续的时间（以秒
   +--------------------------------+-------------------------+----------------------
   |``expires``                     |``None``                 |cookie失效的实际日期/时间。
   +--------------------------------+-------------------------+----------------------
   |``path``                        |``"/"``                  |cookie生效的路径前缀。 浏览
   |                                |                         |
   |                                |                         |当你不是控制你的站点的顶层时
   +--------------------------------+-------------------------+----------------------
   |``domain``                      |``None``                 |这个cookie有效的站点。 你可
   |                                |                         |
   |                                |                         |如果这个参数被设成\ `` Non
   +--------------------------------+-------------------------+----------------------
   |``False``                       |``False``                |如果设置为 ``True`` ，浏览
   +--------------------------------+-------------------------+----------------------
```

好坏参半的Cookies

也许你已经注意到了，cookies的工作方式可能导致的问题。 让我们看一下其中一些比较重要的问题：

> cookie的存储是自愿的，一个客户端不一定要去接受或存储cookie。 事实上，所有的浏览器都让用户自己控制 是否接受cookies。 如果你想知道cookies对于Web应用有多重要，你可以试着打开这个浏览器的 选项：
>
> 尽管cookies广为使用，但仍被认为是不可靠的的。 这意味着，开发者使用cookies之前必须 检查用户是否可以接收cookie。
>
> Cookie(特别是那些没通过HTTPS传输的)是非常不安全的。 因为HTTP数据是以明文发送的，所以 特别容易受到嗅探攻击。 也就是说，嗅探攻击者可以在网络中拦截并读取cookies，因此你要 绝对避免在cookies中存储敏感信息。 这就意味着您不应该使用cookie来在存储任何敏感信息。
>
> 还有一种被称为"中间人"的攻击更阴险，攻击者拦截一个cookie并将其用于另一个用户。 第19章将深入讨论这种攻击的本质以及如何避免。
>
> 即使从预想中的接收者返回的cookie也是不安全的。 在大多数浏览器中您可以非常容易地修改cookies中的信息。有经验的用户甚至可以通过像mechanize(http://wwwsearch.sourceforge.net/mechanize/) 这样的工具手工构造一个HTTP请求。
>
> 因此不能在cookies中存储可能会被篡改的敏感数据。 在cookies中存储 `IsLoggedIn=1` ，以标识用户已经登录。 犯这类错误的站点数量多的令人难以置信； 绕过这些网站的安全系统也是易如反掌。

Django的 Session 框架

由于存在的限制与安全漏洞，cookies和持续性会话已经成为Web开发中令人头疼的典范。 好消息是，Django的目标正是高效的“头疼杀手”，它自带的session框架会帮你搞定这些问题。

你可以用session 框架来存取每个访问者任意数据， 这些数据在服务器端存储，并对cookie的收发进行了抽象。 Cookies只存储数据的哈希会话ID，而不是数据本身，从而避免了大部分的常见cookie问题。

下面我们来看看如何打开session功能，并在视图中使用它。

打开 Sessions功能

Sessions 功能是通过一个中间件(参见第17章)和一个模型(model)来实现的。 要打开sessions功能，需要以下几步操作：

1. 编辑 `MIDDLEWARE_CLASSES` 配置，确保 `MIDDLEWARE_CLASSES` 中包含 `'django.contrib.sessions.middleware.SessionMiddleware'` 。
2. 确认 `INSTALLED_APPS` 中有 `'django.contrib.sessions'` (如果你是刚打开这个应用，别忘了运行 `manage.py syncdb`)

如果项目是用 `startproject` 来创建的，配置文件中都已经安装了这些东西，除非你自己删除，正常情况下，你无需任何设置就可以使用session功能。

如果不需要session功能，你可以删除 `MIDDLEWARE_CLASSES` 设置中的 `SessionMiddleware` 和 `INSTALLED_APPS` 设置中的 `'django.contrib.sessions'` 。虽然这只会节省很少的开销，但积少成多啊。

在视图中使用Session

`SessionMiddleware` 激活后，每个传给视图(view)函数的第一个参数`HttpRequest` 对象都有一个 `session` 属性，这是一个字典型的对象。 你可以象用普通字典一样来用它。 例如，在视图(view)中你可以这样用：

```
# Set a session value:
request.session["fav_color"] = "blue"

# Get a session value -- this could be called in a different view,
# or many requests later (or both):
fav_color = request.session["fav_color"]

# Clear an item from the session:
del request.session["fav_color"]

# Check if the session has a given key:
if "fav_color" in request.session:
    ...
```

其他的映射方法，如 `keys()` 和 `items()` 对 `request.session` 同样有效：

下面是一些有效使用Django sessions的简单规则：

> 用正常的字符串作为key来访问字典 `request.session` ， 而不是整数、对象或其它什么的。
>
> Session字典中以下划线开头的key值是Django内部保留key值。 框架只会用很少的几个下划线 开头的session变量，除非你知道他们的具体含义，而且愿意跟上Django的变化，否则，最好 不要用这些下划线开头的变量，它们会让Django搅乱你的应用。
>
> 比如，不要象这样使用`_fav_color` 会话密钥(session key)：

```
request.session['_fav_color'] = 'blue' # Don't do this!
```

> 不要用一个新对象来替换掉 `request.session` ，也不要存取其属性。 可以像Python中的字典那样使用。 例如：

```
request.session = some_other_object # Don't do this!

request.session.foo = 'bar' # Don't do this!
```

我们来看个简单的例子。 这是个简单到不能再简单的例子：在用户发了一次评论后将 `has_commented` 设置为 `True` 。 这是个简单（但不很安全）的、防止用户多次评论的方法。

```
def post_comment(request):
    if request.method != 'POST':
        raise Http404('Only POSTs are allowed')

    if 'comment' not in request.POST:
        raise Http404('Comment not submitted')

    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")

    c = comments.Comment(comment=request.POST['comment'])
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

下面是一个很简单的站点登录视图(view)：

```
def login(request):
    if request.method != 'POST':
        raise Http404('Only POSTs are allowed')
    try:
        m = Member.objects.get(username=request.POST['username'])
        if m.password == request.POST['password']:
            request.session['member_id'] = m.id
            return HttpResponseRedirect('/you-are-logged-in/')
    except Member.DoesNotExist:
        return HttpResponse("Your username and password didn't match.")
```

下面的例子将登出一个在上面已通过 `login()` 登录的用户：

```
def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
        pass
    return HttpResponse("You're logged out.")
```

注意

在实践中，这是很烂的用户登录方式，稍后讨论的认证(authentication)框架会帮你以更健壮和有利的方式来处理这些问题。 这些非常简单的例子只是想让你知道这一切是如何工作的。这些实例尽量简单，这样你可以更容易看到发生了什么

设置测试Cookies

就像前面提到的，你不能指望所有的浏览器都可以接受cookie。 因此，为了使用方便，Django提供了一个简单的方法来测试用户的浏览器是否接受cookie。 你只需在视图(view)中调用 `request.session.set_test_cookie()`

，并在后续的视图(view)、而不是当前的视图(view)中检查 `request.session.test_cookie_worked()` 。

虽然把 `set_test_cookie()` 和 `test_cookie_worked()` 分开的做法看起来有些笨拙，但由于cookie的工作方式，这无可避免。 当设置一个cookie时候，只能等浏览器下次访问的时候，你才能知道浏览器是否接受cookie。

检查cookie是否可以正常工作后，你得自己用 `delete_test_cookie()` 来清除它，这是个好习惯。 在你证实了测试cookie已工作了之后这样操作。

这是个典型例子：

```
def login(request):

    # If we submitted the form...
    if request.method == 'POST':

        # Check that the test cookie worked (we set it below):
        if request.session.test_cookie_worked():

            # The test cookie worked, so delete it.
            request.session.delete_test_cookie()

            # In practice, we'd need some logic to check username/password
            # here, but since this is an example...
            return HttpResponse("You're logged in.")

        # The test cookie failed, so display an error message. If this
        # were a real site, we'd want to display a friendlier message.
        else:
            return HttpResponse("Please enable cookies and try again.")

    # If we didn't post, send the test cookie along with the login form.
    request.session.set_test_cookie()
    return render_to_response('foo/login_form.html')
```

注意

再次强调，内置的认证函数会帮你做检查的。

在视图(View)外使用Session

从内部来看，每个session都只是一个普通的Django model（在 `django.contrib.sessions.models` 中定义)。每个session都由一个随机的32字节哈希串来标识，并存储于cookie中。 因为它是一个标准的模型，所以你可以使用Django数据库API来存取session。

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

你需要使用 `get_decoded()` 来读取实际的session数据。 这是必需的，因为字典存储为一种特定的编码格式。

```
>>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

何时保存Session

缺省的情况下，Django只会在session发生变化的时候才会存入数据库，比如说，字典赋值或删除。

```
# Session is modified.
request.session['foo'] = 'bar'

# Session is modified.
del request.session['foo']

# Session is modified.
request.session['foo'] = {}

# Gotcha: Session is NOT modified, because this alters
# request.session['foo'] instead of request.session.
request.session['foo']['bar'] = 'baz'
```

你可以设置 `SESSION_SAVE_EVERY_REQUEST` 为 `True` 来改变这一缺省行为。如果置为True的话，Django会在每次收到请求的时候保存session，即使没发生变化。

注意，会话cookie只会在创建和修改的时候才会送出。 但如果 `SESSION_SAVE_EVERY_REQUEST` 设置为 `True` ，会话cookie在每次请求的时候都会送出。 同时，每次会话cookie送出的时候，其 `expires` 参数都会更新。

浏览器关闭即失效会话 vs 持久会话

你可能注意到了，Google给我们发送的cookie中有 `expires=Sun, 17-Jan-2038 19:14:07 GMT;` cookie可以有过期时间，这样浏览器就知道什么时候可以删除cookie了。 如果cookie没有设置过期时间，当用户关闭浏览器的时候，cookie就自动过期了。 你可以改变 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 的设置来控制session框架的这一行为。

缺省情况下， `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置为 `False` ，这样，会话cookie可以在用户浏览器中保持有效达 `SESSION_COOKIE_AGE` 秒（缺省设置是两周，即1,209,600 秒）。 如果你不想用户每次打开浏览器都必须重新登陆的话，用这个参数来帮你。

如果 `SESSION_EXPIRE_AT_BROWSER_CLOSE` 设置为 `True` ，当浏览器关闭时，Django会使cookie失效。

其他的Session设置

除了上面提到的设置，还有一些其他的设置可以影响Django session框架如何使用cookie，详见表 14-2.

表 14-2\. 影响cookie行为的设置

设置	描述	缺省
`SESSION_COOKIE_DOMAIN`	使用会话cookie（session cookies）的站点。 将它设成一个字符串，就好象`` “.example.com”`` 以用于跨站点（cross-domain）的cookie，或`` None`` 以用于单个站点。	`None`
`SESSION_COOKIE_NAME`	会话中使用的cookie的名字。 它可以是任意的字符串。	`"sessionid"`
`SESSION_COOKIE_SECURE`	是否在session中使用安全cookie。 如果设置 `True` , cookie就会标记为安全， 这意味着cookie只会通过HTTPS来传输。	`False`

技术细节

如果你还是好奇的话，下面是一些关于session框架内部工作方式的技术细节：

> session 字典接受任何支持序列化的Python对象。 参考Python内建模块 `pickle` 的文档以获取更多信息。
>
> Session 数据存在数据库表 `django_session` 中
>
> Session 数据在需要的时候才会读取。 如果你从不使用 `request.session` ， Django不会动相关数据库表的一根毛。
>
> Django 只在需要的时候才送出cookie。 如果你压根儿就没有设置任何会话数据，它不会送出会话cookie(除非 `SESSION_SAVE_EVERY_REQUEST` 设置为 `True`)。
>
> Django session 框架完全而且只能基于cookie。 它不会后退到把会话ID编码在URL中（像某些工具(PHP,JSP)那样）。
>
> 这是一个有意而为之的设计。 把session放在URL中不只是难看，更重要的是这让你的站点 很容易受到攻击——通过 `Referer` header进行session ID”窃听”而实施的攻击。

如果你还是好奇，阅读源代码是最直接办法，详见 `django.contrib.sessions` 。

用户与Authentication

通过session，我们可以在多次浏览器请求中保持数据， 接下来的部分就是用session来处理用户登录了。 当然，不能仅凭用户的一面之词，我们就相信，所以我们需要认证。

当然了，Django 也提供了工具来处理这样的常见任务（就像其他常见任务一样）。 Django 用户认证系统处理用户帐号，组，权限以及基于cookie的用户会话。 这个系统一般被称为 *auth/auth* (认证与授权)系统。 这个系统的名称同时也表明了用户常见的两步处理。 我们需要

1. 验证 (认证) 用户是否是他所宣称的用户(一般通过查询数据库验证其用户名和密码)
2. 验证用户是否拥有执行某种操作的 授权 (通常会通过检查一个权限表来确认)

根据这些需求，Django 认证/授权 系统会包含以下的部分：

- 用户：在网站注册的人
- 权限：用于标识用户是否可以执行某种操作的二进制(yes/no)标志
- 组 :一种可以将标记和权限应用于多个用户的常用方法
- *Messages* : 向用户显示队列式的系统消息的常用方法

如果你已经用了admin工具(详见第6章)，就会看见这些工具的大部分。如果你在admin工具中编辑过用户或组，那么实际上你已经编辑过授权系统的数据库表了。

打开认证支持

像session工具一样，认证支持也是一个Django应用，放在 `django.contrib` 中，所以也需要安装。 与session系统相似，它也是缺省安装的，但如果它已经被删除了，通过以下步骤也能重新安装上：

1. 根据本章早前的部分确认已经安装了session 框架。 需要确认用户使用cookie，这样 sesson 框架才能正常使用。
2. 将 `'django.contrib.auth'` 放在你的 `INSTALLED_APPS` 设置中，然后运行 `manage.py syncdb` 以创建对应的数据库表。
3. 确认 `SessionMiddleware` 后面的 `MIDDLEWARE_CLASSES` 设置中包含 `'django.contrib.auth.middleware.AuthenticationMiddleware'` SessionMiddleware。

这样安装后，我们就可以在视图(view)的函数中处理user了。 在视图中存取users，主要用 `request.user` ；这个对象表示当前已登录的用户。 如果用户还没登录，这就是一个 `AnonymousUser` 对象(细节见下)。

你可以很容易地通过 `is_authenticated()` 方法来判断一个用户是否已经登录了：

```
if request.user.is_authenticated():
    # Do something for authenticated users.
else:
    # Do something for anonymous users.
```

使用User对象

`User` 实例一般从 `request.user` ，或是其他下面即将要讨论到的方法取得，它有很多属性和方法。 `AnonymousUser` 对象模拟了 部分 的接口，但不是全部，在把它当成真正的user对象使用前，你得检查一下 `user.is_authenticated()` 表14-3和14-4分别列出了 `User` 对象中的属性（fields)和方法。

表 14-3\. `User` 对象属性

属性	描述
username	必需的，不能多于30个字符。 仅用字母数字式字符（字母、数字和下划线）。
`first_name`	可选; 少于等于30字符。
`last_name`	可选; 少于等于30字符。
`email`	可选。 邮件地址。
`password`	必需的。 密码的哈希值（Django不储存原始密码）。 See the Passwords section for more about this value.
`is_staff`	布尔值。 用户是否拥有网站的管理权限。
`is_active`	布尔值. 设置该账户是否可以登录。 把该标志位置为`False`而不是直接删除账户。
`is_superuser`	布尔值 标识用户是否拥有所有权限，无需显式地权限分配定义。
`last_login`	用户上次登录的时间日期。 它被默认设置为当前的日期/时间。
`date_joined`	账号被创建的日期时间 当账号被创建时，它被默认设置为当前的日期/时间。

System Message: ERROR/3 (<string> , line 735)

Error parsing content block for the “table” directive: exactly one table expected.

```
.. table:: 表 14-4\. ``User`` 对象方法

    +-----------------------------------------------------------------------------------
    |方法
    +===================================================================================
    |``is_authenticated()``
    |
    +-----------------------------------------------------------------------------------
    |``is_anonymous()``
    |
    +-----------------------------------------------------------------------------------
    |``get_full_name()``
    +-----------------------------------------------------------------------------------
    |``set_password(passwd)``
    +-----------------------------------------------------------------------------------
    |check_password(passwd)
    +-----------------------------------------------------------------------------------
    |``get_group_permissions()``
    +-----------------------------------------------------------------------------------
    |``get_all_permissions()``
    +-----------------------------------------------------------------------------------
    |``has_perm(perm)``
    +-----------------------------------------------------------------------------------
    |has_perms(perm_list)
    +-----------------------------------------------------------------------------------
    |``has_module_perms(app_label)``
    +-----------------------------------------------------------------------------------
    |get_and_delete_messages()
    +-----------------------------------------------------------------------------------
    |``email_user(subj, msg)``
    +-----------------------------------------------------------------------------------
```

最后， `User` 对象有两个many-to-many属性。 `groups` 和 `permissions` 。正如其他的many-to-many属性使用的方法一样， `User` 对象可以获得它们相关的对象：

```
# Set a user's groups:
myuser.groups = group_list

# Add a user to some groups:
myuser.groups.add(group1, group2,...)

# Remove a user from some groups:
myuser.groups.remove(group1, group2,...)

# Remove a user from all groups:
myuser.groups.clear()

# Permissions work the same way
myuser.permissions = permission_list
myuser.permissions.add(permission1, permission2, ...)
myuser.permissions.remove(permission1, permission2, ...)
myuser.permissions.clear()
```

登录和退出

Django 提供内置的视图(view)函数用于处理登录和退出 (以及其他奇技淫巧)，但在开始前，我们来看看如何手工登录和退出。 Django提供两个函数来执行 `django.contrib.auth` \中的动作： `authenticate()`

和 `login()` 。

认证给出的用户名和密码，使用 `authenticate()` 函数。它接受两个参数，用户名 username 和 密码 password ，并在密码对给出的用户名合法的情况下返回一个 User 对象。 如果密码不合法， `authenticate()` 返回 `None` 。

```
>>> from django.contrib import auth
>>> user = auth.authenticate(username='john', password='secret')
>>> if user is not None:
...     print "Correct!"
... else:
...     print "Invalid password."
```

`authenticate()` 只是验证一个用户的证书而已。 而要登录一个用户，使用 `login()` 。该函数接受一个 `HttpRequest` 对象和一个 `User` 对象作为参数并使用Django的会话（ `session` ）框架把用户的ID保存在该会话中。

下面的例子演示了如何在一个视图中同时使用 `authenticate()` 和 `login()` 函数：

```
from django.contrib import auth

def login_view(request):
    username = request.POST.get('username', '')
    password = request.POST.get('password', '')
    user = auth.authenticate(username=username, password=password)
    if user is not None and user.is_active:
        # Correct password, and the user is marked "active"
        auth.login(request, user)
        # Redirect to a success page.
        return HttpResponseRedirect("/account/loggedin/")
    else:
        # Show an error page
        return HttpResponseRedirect("/account/invalid/")
```

注销一个用户，在你的视图中使用 `django.contrib.auth.logout()` 。 它接受一个 `HttpRequest` 对象并且没有返回值。

```
from django.contrib import auth

def logout_view(request):
    auth.logout(request)
    # Redirect to a success page.
    return HttpResponseRedirect("/account/loggedout/")
```

注意，即使用户没有登录， `logout()` 也不会抛出任何异常。

在实际中，你一般不需要自己写登录/登出的函数；认证系统提供了一系例视图用来处理登录和登出。 使用认证视图的第一步是把它们写在你的URLconf中。 你需要这样写：

```
from django.contrib.auth.views import login, logout

urlpatterns = patterns('',
    # existing patterns here...
    (r'^accounts/login/$',  login),
    (r'^accounts/logout/$', logout),
)
```

`/accounts/login/` 和 `/accounts/logout/` 是Django提供的视图的默认URL。

缺省情况下， `login` 视图渲染 `registragiton/login.html` 模板(可以通过视图的额外参数 `template_name` 修改这个模板名称)。 这个表单必须包含 `username` 和 `password` 域。如下示例： 一个简单的 template 看起来是这样的

```
{% extends "base.html" %}

{% block content %}

  {% if form.errors %}
    <p class="error">Sorry, that's not a valid username or password</p>
  {% endif %}

  <form action="" method="post">
    <label for="username">User name:</label>
    <input type="text" name="username" value="" id="username">
    <label for="password">Password:</label>
    <input type="password" name="password" value="" id="password">

    <input type="submit" value="login" />
    <input type="hidden" name="next" value="{{ next|escape }}" />
  </form>

{% endblock %}
```

如果用户登录成功，缺省会重定向到 `/accounts/profile` 。 你可以提供一个保存登录后重定向URL的 `next` 隐藏域来重载它的行为。 也可以把值以 `GET` 参数的形式发送给视图函数，它会以变量 `next` 的形式保存在上下文中，这样你就可以把它用在隐藏域上了。

logout视图有一些不同。 默认情况下它渲染 `registration/logged_out.html` 模板（这个视图一般包含你已经成功退出的信息）。 视图中还可以包含一个参数 `next_page` 用于退出后重定向。

限制已登录用户的访问

有很多原因需要控制用户访问站点的某部分。

一个简单原始的限制方法是检查 `request.user.is_authenticated()` ,然后重定向到登陆页面：

```
from django.http import HttpResponseRedirect

def my_view(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/accounts/login/?next=%s' % request.path)
    # ...
```

或者显示一个出错信息：

```
def my_view(request):
    if not request.user.is_authenticated():
        return render_to_response('myapp/login_error.html')
    # ...
```

作为一个快捷方式, 你可以使用便捷的 `login_required` 修饰符:

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    # ...
```

`login_required` 做下面的事情:

- 如果用户没有登录, 重定向到 `/accounts/login/` , 把当前绝对URL作为 `next` 在查询字符串中传递过去, 例如： /accounts/login/?next=/polls/3/ 。
- 如果用户已经登录, 正常地执行视图函数。 视图代码就可以假定用户已经登录了。

对通过测试的用户限制访问

限制访问可以基于某种权限，某些检查或者为login视图提供不同的位置，这些实现方式大致相同。

一般的方法是直接在视图的 `request.user` 上运行检查。 例如，下面视图确认用户登录并是否有 `polls.can_vote` 权限：

```
def vote(request):
    if request.user.is_authenticated() and request.user.has_perm('polls.can_vote')):
        # vote here
    else:
        return HttpResponse("You can't vote in this poll.")
```

并且Django有一个称为 `user_passes_test` 的简洁方式。它接受参数然后为你指定的情况生成装饰器。

```
def user_can_vote(user):
    return user.is_authenticated() and user.has_perm("polls.can_vote")

@user_passes_test(user_can_vote, login_url="/login/")
def vote(request):
    # Code here can assume a logged-in user with the correct permission.
    ...
```

`user_passes_test` 使用一个必需的参数： 一个可调用的方法，当存在 `User` 对象并当此用户允许查看该页面时返回 `True` 。 注意 user_passes_test 不会自动检查 User

是否认证，你应该自己做这件事。

例子中我们也展示了第二个可选的参数 `login_url` ，它让你指定你的登录页面的URL（默认为 `/accounts/login/` ）。 如果用户没有通过测试，那么 `user_passes_test` 将把用户重定向到 `login_url`

既然检查用户是否有一个特殊权限是相对常见的任务，Django为这种情形提供了一个捷径：`permission_required()` 装饰器。 使用这个装饰器，前面的例子可以改写为：

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote', login_url="/login/")
def vote(request):
    # ...
```

注意, `permission_required()` 也有一个可选的 `login_url` 参数, 这个参数默认为 `'/accounts/login/'` 。

限制通用视图的访问

在Django用户邮件列表中问到最多的问题是关于对通用视图的限制性访问。 为实现这个功能，你需要自己包装视图，并且在URLconf中，将你自己的版本替换通用视图：

```
from django.contrib.auth.decorators import login_required
from django.views.generic.date_based import object_detail

@login_required
def limited_object_detail(*args, **kwargs):
    return object_detail(*args, **kwargs)
```

当然, 你可以用任何其他限定修饰符来替换 `login_required` 。

管理 Users, Permissions 和 Groups

管理认证系统最简单的方法是通过管理界面。 第六章讨论了怎样使用Django的管理界面来编辑用户和控制他们的权限和可访问性，并且大多数时间你使用这个界面就可以了。

然而，当你需要绝对的控制权的时候，有一些低层 API 需要深入专研，我们将在下面的章节中讨论它们。

创建用户

使用 `create_user` 辅助函数创建用户:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user(username='john',
...                                 email='jlennon@beatles.com',
...                                 password='glass onion')
```

在这里， `user` 是 `User` 类的一个实例，准备用于向数据库中存储数据。（`create_user()` 实际上没有调用 `save()` ）。 `create_user()` 函数并没有在数据库中创建记录，在保存数据之前，你仍然可以继续修改它的属性值。

```
>>> user.is_staff = True
>>> user.save()
```

修改密码

你可以使用 `set_password()` 来修改密码：

```
>>> user = User.objects.get(username='john')
>>> user.set_password('goo goo goo joob')
>>> user.save()
```

除非你清楚的知道自己在做什么，否则不要直接修改 `password` 属性。 其中保存的是密码的加入*salt*的*hash*值 ，所以不能直接编辑。

一般来说， `User` 对象的 `password` 属性是一个字符串，格式如下：

```
hashtype$salt$hash
```

这是哈希类型，salt和哈希本身，用美元符号（$）分隔。

`hashtype` 是 `sha1` （默认）或者 `md5` ，它是用来处理单向密码哈希的算法。 Salt是一个用来加密原始密码以创建哈希的随机字符串，例如:

```
sha1$a1976$a36cc8cbf81742a8fb52e221aaeab48ed7f58ab4
```

`User.set_password()` 和 `User.check_password()` 函数在后台处理和检查这些值。

salt化得哈希值

一次 哈希 是一次单向的加密过程，你能容易地计算出一个给定值的哈希码，但是几乎不可能从一个哈希码解出它的原值。

如果我们以普通文本存储密码,任何能进入数据库的人都能轻易的获取每个人的密码。 使用哈希方式来存储密码相应的减少了数据库泄露密码的可能。

然而，攻击者仍然可以使用 暴力破解 使用上百万个密码与存储的值对比来获取数据库密码。 这需要花一些时间，但是智能电脑惊人的速度超出了你的想象。

更糟糕的是我们可以公开地得到 *rainbow tables* （一种暴力密码破解表）或预备有上百万哈希密码值的数据库。 使用rainbow tables可以在几秒之内就能搞定最复杂的一个密码。

在存储的hash值的基础上，加入 *salt* 值（一个随机值）， 增加了密码的强度，使得破解更加困难。 因为每个密码的salt值都不相同，这也限制了rainbow table的使用，使得攻击者只能使用最原始的暴力破解方法。

加入salt值得hash并不是绝对安全的存储密码的方法，然而却是安全和方便之间很好的折衷。

处理注册

我们可以使用这些底层工具来创建允许用户注册的视图。 最近每个开发人员都希望实现各自不同的注册方法，所以Django把写注册视图的工作留给了你。 幸运的是，这很容易。

作为这个事情的最简化处理, 我们可以提供一个小视图, 提示一些必须的用户信息并创建这些用户。 Django为此提供了可用的内置表单, 下面这个例子就使用了这个表单:

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response

def register(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            new_user = form.save()
            return HttpResponseRedirect("/books/")
    else:
        form = UserCreationForm()
    return render_to_response("registration/register.html", {
        'form': form,
    })
```

这个表单需要一个叫 `registration/register.html` 的模板。这个模板可能是这样的：

```
{% extends "base.html" %}

{% block title %}Create an account{% endblock %}

{% block content %}
  <h1>Create an account</h1>

  <form action="" method="post">
      {{ form.as_p }}
      <input type="submit" value="Create the account">
  </form>
{% endblock %}
```

在模板中使用认证数据

当前登入的用户以及他（她）的权限可以通过 `RequestContext` 在模板的context中使用（详见第9章）。

注意

从技术上来说，只有当你使用了 `RequestContext` 这些变量才可用。 _并且 _TEMPLATE_CONTEXT_PROCESSORS 设置包含了 “django.core.context_processors.auth” （默认情况就是如此）时， 这些变量才能在模板context中使用。 `TEMPLATE_CONTEXT_PROCESSORS` 设置包含了 `"django.core.context_processors.auth"` （默认情况就是如此）时， 这些变量才能在模板context中使用。

当使用 `RequestContext` 时，当前用户 (是一个 `User` 实例或一个 `AnonymousUser` 实例) 存储在模板变量 `{{ user }}` 中:

```
{% if user.is_authenticated %}
  <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
  <p>Welcome, new user. Please log in.</p>
{% endif %}
```

这些用户的权限信息存储在 `{{ perms }}` 模板变量中。

你有两种方式来使用 `perms` 对象。 你可以使用类似于 `{{ perms.polls }}` 的形式来检查，对于某个特定的应用，一个用户是否具有 任意 权限；你也可以使用 `{{ perms.polls.can_vote }}` 这样的形式，来检查一个用户是否拥有特定的权限。

这样你就可以在模板中的 `{% if %}` 语句中检查权限:

```
{% if perms.polls %}
  <p>You have permission to do something in the polls app.</p>
  {% if perms.polls.can_vote %}
    <p>You can vote!</p>
  {% endif %}
{% else %}
  <p>You don't have permission to do anything in the polls app.</p>
{% endif %}
```

权限、组和消息

在认证框架中还有其他的一些功能。 我们会在接下来的几个部分中进一步地了解它们。

权限

权限可以很方便地标识用户和用户组可以执行的操作。 它们被Django的admin管理站点所使用，你也可以在你自己的代码中使用它们。

Django的admin站点如下使用权限：

- 只有设置了 *add* 权限的用户才能使用添加表单，添加对象的视图。
- 只有设置了 *change* 权限的用户才能使用变更列表，变更表格，变更对象的视图。
- 只有设置了 *delete* 权限的用户才能删除一个对象。

权限是根据每一个类型的对象而设置的，并不具体到对象的特定实例。 例如，我们可以允许Mary改变新故事，但是目前还不允许设置Mary只能改变自己创建的新故事，或者根据给定的状态，出版日期或者ID号来选择权限。

会自动为每一个Django模型创建三个基本权限：增加、改变和删除。 当你运行 `manage.py syncdb` 命令时，这些权限被添加到 `auth_permission` 数据库表中。

权限以 `"<app>.<action>_<object_name>"` 的形式出现。

就跟用户一样，权限也就是Django模型中的 `django.contrib.auth.models` 。因此如果你愿意，你也可以通过Django的数据库API直接操作权限。

组

组提供了一种通用的方式来让你按照一定的权限规则和其他标签将用户分类。 一个用户可以隶属于任何数量的组。

在一个组中的用户自动获得了赋予该组的权限。 例如， `Site editors` 组拥有 `can_edit_home_page` 权限，任何在该组中的用户都拥有这个权限。

组也可以通过给定一些用户特殊的标记，来扩展功能。 例如，你创建了一个 `'Special users'` 组，并且允许组中的用户访问站点的一些VIP部分，或者发送VIP的邮件消息。

和用户管理一样，admin接口是管理组的最简单的方法。 然而，组也就是Django模型 `django.contrib.auth.models` ，因此你可以使用Django的数据库API，在底层访问这些组。

消息

消息系统会为给定的用户接收消息。 每个消息都和一个 `User` 相关联。

在每个成功的操作以后，Django的admin管理接口就会使用消息机制。 例如，当你创建了一个对象，你会在admin页面的顶上看到 `The object was created successfully` 的消息。

你也可以使用相同的API在你自己的应用中排队接收和显示消息。 API非常地简单：

- 要创建一条新的消息，使用 `user.message_set.create(message='message_text')` 。
- 要获得/删除消息，使用 `user.get_and_delete_messages()` ，这会返回一个 `Message` 对象的列表，并且从队列中删除返回的项。

在例子视图中，系统在创建了播放单（playlist）以后，为用户保存了一条消息。

```
def create_playlist(request, songs):
    # Create the playlist with the given songs.
    # ...
    request.user.message_set.create(
        message="Your playlist was added successfully."
    )
    return render_to_response("playlists/create.html",
        context_instance=RequestContext(request))
```

当使用 `RequestContext` ，当前登录的用户以及他（她）的消息，就会以模板变量 `{{ messages }}` 出现在模板的context中。

```
{% if messages %}
<ul>
    {% for message in messages %}
    <li>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

需要注意的是 `RequestContext` 会在后台调用 `get_and_delete_messages` ，因此即使你没有显示它们，它们也会被删除掉。

最后注意，这个消息框架只能服务于在用户数据库中存在的用户。 如果要向匿名用户发送消息，请直接使用会话框架。

下一章

是的，会话和认证系统有太多的东西要学。 大多数情况下，你并不需要本章所提到的所有功能。

在下一章，我们会看一下Django的缓存机制，这是一个提高你的网页应用性能的便利的办法。

第十五章： 缓存机制

动态网站的问题就在于它是动态的。 也就是说每次用户访问一个页面，服务器要执行数据库查询，启动模板，执行业务逻辑以及最终生成一个你所看到的网页，这一切都是动态即时生成的。 从处理器资源的角度来看，这是比较昂贵的。

对于大多数网络应用来说，过载并不是大问题。 因为大多数网络应用并不是washingtonpost.com或Slashdot；它们通常是很小很简单，或者是中等规模的站点，只有很少的流量。 但是对于中等至大规模流量的站点来说，尽可能地解决过载问题是非常必要的。

这就需要用到缓存了。

缓存的目的是为了避免重复计算，特别是对一些比较耗时间、资源的计算。 下面的伪代码演示了如何对动态页面的结果进行缓存。

```
given a URL, try finding that page in the cache
if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

为此，Django提供了一个稳定的缓存系统让你缓存动态页面的结果，这样在接下来有相同的请求就可以直接使用缓存中的数据，避免不必要的重复计算。 另外Django还提供了不同粒度数据的缓存，例如： 你可以缓存整个页面，也可以缓存某个部分，甚至缓存整个网站。

Django也和"上游"缓存工作的很好，例如Squid(http://www.squid-cache.org)和基于浏览器的缓存。 这些类型的缓存你不直接控制，但是你可以提供关于你的站点哪部分应该被缓存和怎样缓存的线索(通过HTTP头部)给它们

设定缓存

缓存系统需要一些少量的设定工作。 也就是说，你必须告诉它缓存的数据应该放在哪里，在数据库中，在文件系统，或直接在内存中。 这是一个重要的决定，影响您的高速缓存的性能，是的，有些类型的缓存比其它类型快。

缓存设置在settings文件的 `CACHE_BACKEND` 中。 这里是一个CACHE_BACKEND所有可用值的解释。

内存缓冲

Memcached是迄今为止可用于Django的最快，最有效的缓存类型，Memcached是完全基于内存的缓存框架，最初开发它是用以处理高负荷的LiveJournal.com随后由Danga Interactive公司开源。 它被用于一些站点，例如Facebook和维基百科网站，以减少数据库访问，并大幅提高站点的性能。

Memcached是免费的（http://danga.com/memcached）。它作为一个守护进程运行，并分配了特定数量的内存。 它只是提供了添加，检索和删除缓存中的任意数据的高速接口。 所有数据都直接存储在内存中，所以没有对使用的数据库或文件系统的开销。

在安装了Memcached本身之后，你将需要安装Memcached Python绑定，它没有直接和Django绑定。 这两个可用版本。 选择和安装以下模块之一：

- 最快的可用选项是一个模块，称为cmemcache，在http://gijsbert.org/cmemcache。
- 如果您无法安装cmemcache，您可以安装python - Memcached，在ftp://ftp.tummy.com/pub/python-memcached/。如果该网址已不再有效，只要到Memcached的网站http://www.danga.com/memcached/），并从客户端API完成Python绑定。

若要使用Memcached的Django，设置CACHE_BACKEND到memcached：/ / IP：port/，其中IP是Memcached的守护进程的IP地址，port是Memcached运行的端口。

在这个例子中，Memcached运行在本地主机 (127.0.0.1)上,端口为11211：

```
CACHE_BACKEND = 'memcached://127.0.0.1:11211/'
```

Memcached的一个极好的特性是它在多个服务器间分享缓存的能力。 这意味着您可以在多台机器上运行Memcached的守护进程，该程序会把这些机器当成一个单一缓存，而无需重复每台机器上的缓存值。 要充分利用此功能，请在CACHE_BACKEND里引入所有服务器的地址，用分号分隔。

这个例子中，缓存在运行在IP地址为172.19.26.240和172.19.26.242，端口号为11211的Memcached实例间分享:

```
CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11211/'
```

这个例子中，缓存在运行在172.19.26.240(端口11211)，172.19.26.242(端口11212)，172.19.26.244(端口11213)的Memcached实例间分享:

```
CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11212;172.19.26.244:11213/
```

最后有关Memcached的一点是，基于内存的缓存有一个重大的缺点。 由于缓存的数据存储在内存中，所以如果您的服务器崩溃，数据将会消失。 显然，内存不是用来持久化数据的，因此不要把基于内存的缓存作为您唯一的存储数据缓存。 毫无疑问，在Django的缓存后端不应该用于持久化，它们本来就被设计成缓存的解决方案。但我们仍然指出此点，这里是因为基于内存的缓存是暂时的。

数据库缓存

为了使用数据库表作为缓存后端，首先在数据库中运行这个命令以创建缓存表：

```
python manage.py createcachetable [cache_table_name]
```

这里的[cache_table_name]是要创建的数据库表名。 （这个名字随你的便，只要它是一个有效的表名，而且不是已经在您的数据库中使用的表名。）这个命令以Django的数据库缓存系统所期望的格式创建一个表。

一旦你创建了数据库表，把你的CACHE_BACKEND设置为"db://tablename"，这里的tablename是数据库表的名字，在这个例子中，缓存表名为my_cache_table: 在这个例子中，高速缓存表的名字是my_cache_table：

```
CACHE_BACKEND = 'db://my_cache_table'
```

数据库缓存后端使用你的settings文件指定的同一数据库。 你不能为你的缓存表使用不同的数据库后端.

如果你已经有了一个快速，良好的索引数据库服务器，那么数据库缓存的效果最明显。

文件系统缓存

要把缓存项目放在文件系统上，请为CACHE_BACKEND使用"file://"的缓存类型。例如，要把缓存数据存储在/var/tmp/django_cache上，请使用此设置：

```
CACHE_BACKEND = 'file:///var/tmp/django_cache'
```

注意例子中开头有三个斜线。 头两项是file://，第三个是第一个字符的目录路径，/var/tmp/django_cache。如果你使用的是Windows，在file://之后加上文件的驱动器号：

```
file://c:/foo/bar
```

目录路径应该是绝对路径，即应该以你的文件系统的根开始。 在设置的结尾放置斜线与否无关紧要。

确认该设置指向的目录存在并且你的Web服务器运行的系统的用户可以读写该目录。 继续上面的例子，如果你的服务器以用户apache运行，确认/var/tmp/django_cache存在并且用户apache可以读写/var/tmp/django_cache目录。

每个缓存值将被存储为单独的文件，其内容是Python的pickle模块以序列化(“pickled”)形式保存的缓存数据。 每个文件的名称是缓存键，以规避开安全文件系统的使用。

本地内存缓存

如果你想利用内存缓存的速度优势，但又不能使用Memcached，可以考虑使用本地存储器缓存后端。 此缓存的多进程和线程安全。 设置 `CACHE_BACKEND` 为 `locmem:///` 来使用它，例如:

```
CACHE_BACKEND = 'locmem:///'
```

请注意，每个进程都有自己私有的缓存实例，这意味着跨进程缓存是不可能的。 这显然也意味着本地内存缓存效率并不是特别高，所以对产品环境来说它可能不是一个好选择。 对开发来说还不错。

仿缓存（供开发时使用）

最后，Django提供了一个假缓存（只是实现了缓存接口，实际上什么都不做）。

假如你有一个产品站点，在许多地方使用高度缓存，但在开发/测试环境中，你不想缓存，也不想改变代码，这就非常有用了。 要激活虚拟缓存，就像这样设置CACHE_BACKEND：

```
CACHE_BACKEND = 'dummy:///'
```

使用自定义缓存后端

尽管Django包含对许多缓存后端的支持，在某些情况下，你仍然想使用自定义缓存后端。 要让Django使用外部缓存后端，需要使用一个Python import路径作为的CACHE_BACKEND URI的（第一个冒号前的部分），像这样：

```
CACHE_BACKEND = 'path.to.backend://'
```

如果您构建自己的后端，你可以参考标准缓存后端的实现。 源代码在Django的代码目录的django/core/cache/backends/下。

注意 如果没有一个真正令人信服的理由，比如主机不支持，你就应该坚持使用Django包含的缓存后端。 它们经过大量测试，并且易于使用。

CACHE_BACKEND参数

每个缓存后端都可能使用参数。 它们在CACHE_BACKEND设置中以查询字符串形式给出。 有效参数如下：

> `timeout` :用于缓存的过期时间，以秒为单位。 这个参数默认被设置为300秒（五分钟）。
>
> max_entries：对于内存，文件系统和数据库后端，高速缓存允许的最大条目数，超出这个数则旧值将被删除。 这个参数默认是300。
>
> `cull_percentage` :当达到 `max_entries` 的时候,被删除的条目比率。 实际的比率是 `1/cull_percentage` ,所以设置cull_frequency=2就是在达到 `max_entries` 的时候去除一半数量的缓存。
>
> 把 `cull_frequency` 的值设置为 `0` 意味着当达到 `max_entries` 时,缓存将被清空。 这将以很多缓存丢失为代价,大大提高接受访问的速度。

在这个例子中， `timeout` 被设成 `60`

```
CACHE_BACKEND = "memcached://127.0.0.1:11211/?timeout=60"
```

而在这个例子中， `timeout` 设为 `30` 而 `max_entries` 为 `400` ：

```
CACHE_BACKEND = "locmem:///?timeout=30&max_entries=400"
```

其中，非法的参数与非法的参数值都将被忽略。

站点级 Cache

一旦高速缓存设置，最简单的方法是使用缓存缓存整个网站。 您 需要添加'django.middleware.cache.UpdateCacheMiddleware'和'django.middleware.cache.FetchFromCacheMiddleware'到您的MIDDLEWARE_CLASSES设置中，在这个例子中是：

```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
)
```

注意：

不，这里并没有排版错误：修改的中间件，必须放在列表的开始位置，而fectch中间件，必须放在最后。细节有点费解，如果您想了解完整内幕请参看下面的MIDDLEWARE_CLASSES顺序。

然后，在你的Django settings文件里加入下面所需的设置：

- `CACHE_MIDDLEWARE_SECONDS` :每个页面应该被缓存的秒数。
- `CACHE_MIDDLEWARE_KEY_PREFIX` ：如果缓存被多个使用相同Django安装的网站所共享，那么把这个值设成当前网站名，或其他能代表这个Django实例的唯一字符串，以避免key发生冲突。如果你不在意的话可以设成空字符串。

缓存中间件缓存每个没有GET或者POST参数的页面。或者，如果CACHE_MIDDLEWARE_ANONYMOUS_ONLY设置为True，只有匿名请求（即不是由登录的用户）将被缓存。如果想取消用户相关页面（user-specific pages）的缓存，例如Djangos的管理界面，这是一种既简单又有效的方法。CACHE_MIDDLEWARE_ANONYMOUS_ONLY，你应该确保你已经启动AuthenticationMiddleware。

此外，缓存中间件为每个HttpResponse自动设置了几个头部信息：

- 当一个新(没缓存的)版本的页面被请求时设置Last-Modified头部为当前日期/时间。
- 设置Expires头部为当前日期/时间加上定义的CACHE_MIDDLEWARE_SECONDS。
- 设置Cache-Control头部来给页面一个最长的有效期，值来自于CACHE_MIDDLEWARE_SECONDS设置。

参阅更多的中间件第17章。

如果视图设置自己的缓存到期时间（即 它有一个最大年龄在头部信息的`Cache-Control`中），那么页面将缓存直到过期，而不是CACHE_MIDDLEWARE_SECONDS。使用django.views.decorators.cache装饰器，您可以轻松地设置视图的到期时间（使用cache_control装饰器）或禁用缓存视图（使用never_cache装饰器）。请参阅下面的"使用其他头部信息"小节以了解装饰器的更多信息。

视图级缓存

更加颗粒级的缓存框架使用方法是对单个视图的输出进行缓存。`django.views.decorators.cache`定义了一个自动缓存视图响应的`cache_page`装饰器。他是很容易使用的:

```
from django.views.decorators.cache import cache_page

def my_view(request):
    # ...

my_view = cache_page(my_view, 60 * 15)
```

也可以使用Python2.4的装饰器语法：

```
@cache_page(60 * 15)
def my_view(request):
    # ...
```

`cache_page` 只接受一个参数： 以秒计的缓存超时时间。 在前例中， “my_view()” 视图的结果将被缓存 15 分钟。 （注意： 为了提高可读性，该参数被书写为 `60 * 15` 。 `60 * 15` 将被计算为 `900` ， 也就是说15 分钟乘以每分钟 60 秒。）

和站点缓存一样，视图缓存与 URL 无关。 如果多个 URL 指向同一视图，每个视图将会分别缓存。 继续 `my_view` 范例，如果 URLconf 如下所示：

```
urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', my_view),
)
```

那么正如你所期待的那样，发送到 `/foo/1/` 和 `/foo/23/` 的请求将会分别缓存。 但一旦发出了特定的请求（如： /foo/23/ ）， 之后再度发出的指向该 URL 的请求将使用缓存。

在 URLconf 中指定视图缓存

前一节中的范例将视图硬编码为使用缓存，因为 `cache_page` 在适当的位置对 `my_view` 函数进行了转换。 该方法将视图与缓存系统进行了耦合，从几个方面来说并不理想。 例如，你可能想在某个无缓存的站点中重用该视图函数，或者你可能想将该视图发布给那些不想通过缓存使用它们的人。 解决这些问题的方法是在 URLconf 中指定视图缓存，而不是紧挨着这些视图函数本身来指定。

完成这项工作非常简单： 在 URLconf 中用到这些视图函数的时候简单地包裹一个 `cache_page` 。以下是刚才用到过的 URLconf : 这是之前的URLconf :

```
urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', my_view),
)
```

以下是同一个 URLconf ， 不过用 `cache_page` 包裹了 `my_view` ：

```
from django.views.decorators.cache import cache_page

urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', cache_page(my_view, 60 * 15)),
)
```

如果采取这种方法, 不要忘记在 URLconf 中导入 `cache_page` 。

模板碎片缓存

你同样可以使用 `cache` 标签来缓存模板片段。 在模板的顶端附近加入 `{% load cache %}` 以通知模板存取缓存标签。

模板标签 `{% cache %}` 在给定的时间内缓存了块的内容。 它至少需要两个参数: 缓存超时时间（以秒计）和指定缓存片段的名称。 示例：

```
{% load cache %}
{% cache 500 sidebar %}
    .. sidebar ..
{% endcache %}
```

有时你可能想缓存基于片段的动态内容的多份拷贝。 比如，你想为上一个例子的每个用户分别缓存侧边栏。 这样只需要给 `{% cache %}` 传递额外的参数以标识缓存片段。

```
{% load cache %}
{% cache 500 sidebar request.user.username %}
    .. sidebar for logged in user ..
{% endcache %}
```

传递不止一个参数也是可行的。 简单地把参数传给 `{% cache %}` 。

缓存超时时间可以作为模板变量，只要它可以解析为整数值。 例如，如果模板变量 `my_timeout` 值为600，那么以下两个例子是等价的。

```
{% cache 600 sidebar %} ... {% endcache %}
{% cache my_timeout sidebar %} ... {% endcache %}
```

这个特性在避免模板重复方面非常有用。 可以把超时时间保存在变量里，然后在别的地方复用。

低层次缓存API

有些时候，对整个经解析的页面进行缓存并不会给你带来太多好处，事实上可能会过犹不及。

比如说，也许你的站点所包含的一个视图依赖几个费时的查询，每隔一段时间结果就会发生变化。 在这种情况下，使用站点级缓存或者视图级缓存策略所提供的整页缓存并不是最理想的，因为你可能不会想对整个结果进行缓存（因为一些数据经常变化），但你仍然会想对很少变化的部分进行缓存。

针对这样的情况，Django提供了简单低级的缓存API。 你可以通过这个API，以任何你需要的粒度来缓存对象。 你可以对所有能够安全进行 pickle 处理的 Python 对象进行缓存： 字符串、字典和模型对象列表等等。 （查阅 Python 文档可以了解到更多关于 pickling 的信息。）

缓存模块 `django.core.cache` 拥有一个自动依据 `CACHE_BACKEND` 设置创建的 `django.core.cache` 对象。

```
>>> from django.core.cache import cache
```

基本的接口是 `set(key, value, timeout_seconds)` 和 `get(key)` ：

```
>>> cache.set('my_key', 'hello, world!', 30)
>>> cache.get('my_key')
'hello, world!'
```

`timeout_seconds` 参数是可选的, 并且默认为前面讲过的 `CACHE_BACKEND` 设置中的 `timeout` 参数.

如果缓存中不存在该对象，那么 `cache.get()` 会返回 `None` 。

```
# Wait 30 seconds for 'my_key' to expire...

>>> cache.get('my_key')
None
```

我们不建议在缓存中保存 `None` 常量，因为你将无法区分你保存的 `None` 变量及由返回值 `None` 所标识的缓存未命中。

`cache.get()` 接受一个 `缺省` 参数。 它指定了当缓存中不存在该对象时所返回的值：

```
>>> cache.get('my_key', 'has expired')
'has expired'
```

使用 `add()` 方法来新增一个原来没有的键值。 它接受的参数和 `set()` 一样，但是并不去尝试更新已经存在的键值。

```
>>> cache.set('add_key', 'Initial value')
>>> cache.add('add_key', 'New value')
>>> cache.get('add_key')
'Initial value'
```

如果想确定 `add()` 是否成功添加了缓存值，你应该测试返回值。 成功返回True，失败返回False。

还有个 `get_many()` 接口。 `get_many()` 所返回的字典包括了你所请求的存在于缓存中且未超时的所有键值。

```
>>> cache.set('a', 1)
>>> cache.set('b', 2)
>>> cache.set('c', 3)
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

最后,你可以用 `cache.delete()` 显式地删除关键字。

```
>>> cache.delete('a')
```

也可以使用 `incr()` 或者 `decr()` 来增加或者减少已经存在的键值。 默认情况下，增加或减少的值是1。可以用参数来制定其他值。 如果尝试增减不存在的键值会抛出ValueError。

```
>>> cache.set('num', 1)
>>> cache.incr('num')
2
>>> cache.incr('num', 10)
12
>>> cache.decr('num')
11
>>> cache.decr('num', 5)
6
```

注意

`incr()` / `decr()` 方法不是原子操作。 在支持原子增减的缓存后端上（最著名的是memcached），增减操作才是原子的。 然而，如果后端并不原生支持增减操作，也可以通过取值/更新两步操作来实现。

上游缓存

目前为止，本章的焦点一直是对你 自己的 数据进行缓存。 但还有一种与 Web 开发相关的缓存： 上游缓存。 有一些系统甚至在请求到达站点之前就为用户进行页面缓存。

下面是上游缓存的几个例子：

- 你的 ISP (互联网服务商)可能会对特定的页面进行缓存，因此如果你向 http://example.com/ 请求一个页面，你的 ISP 可能无需直接访问 example.com 就能将页面发送给你。 而 example.com 的维护者们却无从得知这种缓存，ISP 位于 example.com 和你的网页浏览器之间，透明地处理所有的缓存。

- 你的 Django 网站可能位于某个 代理缓存 之后，例如 Squid 网页代理缓存 (http://www.squid-cache.org/)，该缓存为提高性能而对页面进行缓存。 在此情况下 ，每个请求将首先由代理服务器进行处理，然后仅在需要的情况下才被传递至你的应用程序。
- 你的网页浏览器也对页面进行缓存。 如果某网页送出了相应的头部，你的浏览器将在为对该网页的后续的访问请求使用本地缓存的拷贝，甚至不会再次联系该网页查看是否发生了变化。

上游缓存将会产生非常明显的效率提升，但也存在一定风险。 许多网页的内容依据身份验证以及许多其他变量的情况发生变化，缓存系统仅盲目地根据 URL 保存页面，可能会向这些页面的后续访问者暴露不正确或者敏感的数据。

举个例子，假定你在使用网页电邮系统，显然收件箱页面的内容取决于登录的是哪个用户。 如果 ISP 盲目地缓存了该站点，那么第一个用户通过该 ISP 登录之后，他（或她）的用户收件箱页面将会缓存给后续的访问者。 这一点也不好玩。

幸运的是， HTTP 提供了解决该问题的方案。 已有一些 HTTP 头标用于指引上游缓存根据指定变量来区分缓存内容，并通知缓存机制不对特定页面进行缓存。 我们将在本节后续部分将对这些头标进行阐述。

使用 Vary头部

`Vary` 头部定义了缓存机制在构建其缓存键值时应当将哪个请求头标考虑在内。 例如，如果网页的内容取决于用户的语言偏好，该页面被称为根据语言而不同。

缺省情况下，Django 的缓存系统使用所请求的路径（比如：`"/stories/2005/jun/23/bank_robbed/"` ）来创建其缓存键。这意味着每次请求都会使用同样的缓存版本，不考虑才客户端cookie和语言配置的不同。 除非你使用 `Vary` 头部通知缓存机制页面输出要依据请求头里的cookie，语言等的设置而不同。

要在 Django 完成这项工作，可使用便利的 `vary_on_headers` 视图装饰器，如下所示：

```
from django.views.decorators.vary import vary_on_headers

# Python 2.3 syntax.
def my_view(request):
    # ...
my_view = vary_on_headers(my_view, 'User-Agent')

# Python 2.4+ decorator syntax.
@vary_on_headers('User-Agent')
def my_view(request):
    # ...
```

在这种情况下，缓存机制（如 Django 自己的缓存中间件）将会为每一个单独的用户浏览器缓存一个独立的页面版本。

使用 `vary_on_headers` 装饰器而不是手动设置 `Vary` 头部（使用像 `response['Vary'] = 'user-agent'` 之类的代码）的好处是修饰器在（可能已经存在的） *Vary* 之上进行 `添加` ，而不是从零开始设置，且可能覆盖该处已经存在的设置。

你可以向 `vary_on_headers()` 传入多个头标：

```
@vary_on_headers('User-Agent', 'Cookie')
def my_view(request):
    # ...
```

该段代码通知上游缓存对 两者 都进行不同操作，也就是说 user-agent 和 cookie 的每种组合都应获取自己的缓存值。 举例来说，使用 `Mozilla` 作为 user-agent 而 `foo=bar` 作为 cookie 值的请求应该和使用 `Mozilla` 作为 user-agent 而 `foo=ham` 的请求应该被视为不同请求。

由于根据 cookie 而区分对待是很常见的情况，因此有 `vary_on_cookie` 装饰器。 以下两个视图是等效的：

```
@vary_on_cookie
def my_view(request):
    # ...

@vary_on_headers('Cookie')
def my_view(request):
    # ...
```

传入 `vary_on_headers` 头标是大小写不敏感的； `"User-Agent"` 与 `"user-agent"` 完全相同。

你也可以直接使用帮助函数： `django.utils.cache.patch_vary_headers` 。 该函数设置或增加 Vary header ， 例如：

```
from django.utils.cache import patch_vary_headers

def my_view(request):
    # ...
    response = render_to_response('template_name', context)
    patch_vary_headers(response, ['Cookie'])
    return response
```

`patch_vary_headers` 以一个 `HttpResponse` 实例为第一个参数，以一个大小写不敏感的头标名称列表或元组为第二个参数。

控制缓存： 使用其它头部

关于缓存剩下的问题是数据的隐私性以及在级联缓存中数据应该在何处储存的问题。

通常用户将会面对两种缓存： 他或她自己的浏览器缓存（私有缓存）以及他或她的提供者缓存（公共缓存）。 公共缓存由多个用户使用，而受其他某人的控制。 这就产生了你不想遇到的敏感数据的问题，比如说你的银行账号被存储在公众缓存中。 因此，Web 应用程序需要以某种方式告诉缓存那些数据是私有的，哪些是公共的。

解决方案是标示出某个页面缓存应当是私有的。 要在 Django 中完成此项工作，可使用 `cache_control` 视图修饰器： 例如：

```
from django.views.decorators.cache import cache_control

@cache_control(private=True)
def my_view(request):
    # ...
```

该修饰器负责在后台发送相应的 HTTP 头部。

还有一些其他方法可以控制缓存参数。 例如, HTTP 允许应用程序执行如下操作:

- 定义页面可以被缓存的最大时间。
- 指定某个缓存是否总是检查较新版本，仅当无更新时才传递所缓存内容。 （一些缓存即便在服务器页面发生变化的情况下仍然会传送所缓存的内容，只因为缓存拷贝没有过期。）

在 Django 中，可使用 `cache_control` 视图修饰器指定这些缓存参数。 在本例中， `cache_control` 告诉缓存对每次访问都重新验证缓存并在最长 3600 秒内保存所缓存版本：

```
from django.views.decorators.cache import cache_control

@cache_control(must_revalidate=True, max_age=3600)
def my_view(request):
    # ...
```

在 `cache_control()` 中，任何合法的 `Cache-Control` HTTP 指令都是有效的。下面是完整列表：

- `public=True`
- `private=True`
- `no_cache=True`
- `no_transform=True`
- `must_revalidate=True`
- `proxy_revalidate=True`
- `max_age=num_seconds`

- `s_maxage=num_seconds`

缓存中间件已经使用 `CACHE_MIDDLEWARE_SETTINGS` 设置设定了缓存头部 max-age 。 如果你在cache_control修饰器中使用了自定义的max_age，该修饰器将会取得优先权，该头部的值将被正确地被合并。

如果你想用头部完全禁掉缓存， `django.views.decorators.cache.never_cache` 装饰器可以添加确保响应不被缓存的头部信息。 例如：

```
from django.views.decorators.cache import never_cache

@never_cache
def myview(request):
    # ...
```

其他优化

Django 带有一些其它中间件可帮助您优化应用程序的性能:

- `django.middleware.http.ConditionalGetMiddleware` 为现代浏览器增加了有条件的，基于 `ETag` 和 `Last-Modified` 头标的GET响应的相关支持。
- `django.middleware.gzip.GZipMiddleware` 为所有现代浏览器压缩响应内容，以节省带宽和传送时间。

MIDDLEWARE_CLASSES 的顺序

如果使用缓存中间件，注意在 `MIDDLEWARE_CLASSES` 设置中正确配置。 因为缓存中间件需要知道哪些头部信息由哪些缓存区来区分。 中间件总是尽可能得想 `Vary` 响应头中添加信息。

`UpdateCacheMiddleware` 在相应阶段运行。因为中间件是以相反顺序运行的，所有列表顶部的中间件反而*last*在相应阶段的最后运行。 所有，你需要确保 `UpdateCacheMiddleware` 排在任何可能往*Vary*头部添加信息的中间件 `之前` 。 下面的中间件模块就是这样的：

- 添加 `Cookie` 的 `SessionMiddleware`
- 添加 `Accept-Encoding` 的 `GZipMiddleware`
- 添加 `Accept-Language` 的 `LocaleMiddleware`

另一方面， `FetchFromCacheMiddleware` 在请求阶段运行，这时中间件循序执行，所以列表顶端的项目会首先执行。 `FetchFromCacheMiddleware` 也需要在会修改 `Vary` 头部的中间件之后运行，所以 `FetchFromCacheMiddleware` 必须放在它们后面。

下一章

Django捆绑了一系列可选的方便特性。 我们已经介绍了一些： admin站点（第六章）和session/user框架（第十四章）。 下一章中，我们将讲述Django中其他的子框架。

第十六章：集成的子框架django.contrib

Python有众多优点，其中之一就是“开机即用”原则： 安装Python的同时会安装好大量的标准软件包，这样 你可以立即使用而不用自己去下载?Django也遵循这个原则，它同样包含了自己的标准库?这一章就来讲 这些集成的子框架

Django标准

Django的标准库存放? `django.contrib` 包中。每个子包都是一个独立的附加功能包?这些子包一般是互相独立的，不过有些 `django.contrib` 子包需要依赖其他子包

? `django.contrib` 中对函数的类型并没有强制要求 。其中一些包中带有模型（因此需要你在数据库中安装对应的数据表），但其它一些由独立的中间件及模板标签组成

`django.contrib` 开发包共有的特性是: 就算你将整个 `django.contrib` 开发包删除，你依然可以使用 Django 的基础功能而不会遇到任何问题??Django 开发者向框架增加新功能的时，他们会严格根据这一原则来决定是否把新功能放?tt class="docutils literal">django.contrib中

`django.contrib` 由以下开发包组成

- `admin` ：自动化的站点管理工具?请查看第6章
- `admindocs` :为Django admin站点提供自动文档?本书没有介绍这方面的知识；详情请参阅Django官方文档
- `auth` ：Django的用户验证框架?参见第十四章
- `comments` ：一个评论应用，目前，这个应用正在紧张的开发中，因此在本书出版的时候还不能给出一个完整的说明，关于这个应用的更多信息请参见Django的官方网? 本书没有介绍这方面的知识；详情请参阅Django官方文档
- `contenttypes` ：这是一个用于引入文档类型的框架，每个安装的Django模块作为一种独立的文档类型?这个框架主要在Django内部被其他应用使用，它主要面向Django的高级开发者?可以通过阅读源码来了解关于这个框架的更多信息，源码的位置? `django/contrib/contenttypes/`
- `csrf` ：这个模块用来防御跨站请求伪?CSRF)。参 见后面标题为”CSRF 防御”的小节
- `databrowse` ：帮助你浏览数据的Django应用?本书没有介绍这方面的知识；详情请参阅Django官方文档
- `flatpages` ：一个在数据库中管理单一HTML内容的模块?参见后面标题为“Flatpages”的小节

- `formtools` ：一些列处理表单通用模式的高级库?本书没有介绍这方面的知识；详情请参阅Django官方文档
- `gis` ：为Django提供GIS（Geographic Information Systems）支持的扩展?举个例子，它允许你的Django模型保存地理学数据并执行地理学查询?这个库比较复杂，本书不详细介绍?请参?a class="reference external" href="javascript:if(confirm('http://geodjango.org/ \n\nţδ Teleport Pro ȡĩÏ ·'ǭķX \n\nÇɼϴ'))window.location='http://geodjango.org/'" tppabs="http://geodjango.org/">http://geodjango.org/上的文档
- `humanize` ：一系列 Django 模块过滤器，用于增加数据的人性化?参阅稍后的章节《人性化数据》
- `localflavor` ：针对不同国家和文化的混杂代码段?例如，它包含了验证美国的邮编 以及爱尔兰的身份证号的方法
- `markup` ：一系列?Django 模板过滤器，用于实现一些常用标记语言?参阅后续章节《标记过滤器》
- `redirects` ：用来管理重定向的框架?参看后面的“重定向”小节
- `sessions` ：Django 的会话框架?参见14章
- `sitemaps` ：用来生成网站地图?XML 文件的框架?参见13章
- `sites` ：一个让你可以在同一个数据库?Django 安装中管理多个网站的框架?参见下一节：
- `syndication` ：一个用 RSS ?Atom 来生成聚合订阅源的的框架?参见13章
- `webdesign` ：对设计者非常有用的Django扩展?到编写此文时，它只包含一个模板标?tt class="docutils literal">{% lorem %}。详情参阅Django文档

本章接下来将详细描述前面没有介绍过的 `django.contrib` 开发包内容

多个站点

Django 的多站点系统是一种通用框架，它让你可以在同一个数据库和同一个Django项目下操作多个网站?这是一个抽象概念，理解起来可能有点困难，因此我们从几个让它能派上用场的实际情景入手

情景1：多站点间复用数

正如我们在第一章里所讲，Django 构建的网?LJWorld.com ?Lawrance.com 是用由同一个新闻组织控制的?肯萨斯州劳伦斯市?劳伦斯日报世 报纸*?LJWorld.com* 主要做新闻，*?Lawrence.com* 关注本地娱乐?然而有时，编辑可能需要把一篇文章发布到 两个 网站上

解决此问题的死脑筋方法可能是使用每个站点分别使用不同的数据库，然后要求站点维护者把同一篇文章发布两次： 一次为 *LJWorld.com*， 另一次为*Lawrence.com?*但这对站点管理员来说是低效率的，而且为同一篇文章在数据库里保留多个副本也显得多余

更好的解决方案？ 两个网站用的是同一个文章数据库，并将每一篇文章与一个或多个站点用多对多关系关联起来*?Django* 站点框架提供数据库表来记载哪些文章可以被关联*?*它是一个把数据与一个或多个站点关联起来的钩子

情景*2*：把网站的名*?*域名保存在一个地

LJWorld.com ?Lawrence.com 都有邮件提醒功能，使读者注册后可以在新闻发生后立即收到通知*?*这是一种完美的的机制： 某读者提交了注册表单，然后马上就受到一封内容是"感谢您的注册"的邮件

把这个注册过程的代码实现两遍显然是低效、多余的，因此两个站点在后台使用相同的代码*?*但感谢注册的通知在两个网站中需要不同*?*通过使用 `Site` 对象，我们通过使用当前站点*?* `name` *(*例如 `'LJWorld.com'` *)?* `domain` *(*例如 `'www.ljworld.com'` *)*可以把感谢通知抽提出来

Django 的多站点框架为你提供了一个位置来存储 *Django* 项目中每个站点的 `name` *?* `domain` ，这意味着你可以用同样的方法来重用这些值

如何使用多站点框

多站点框架与其说是一个框架，不如说是一系列约定*?*所有的一切都基于两个简单的概念

- 位于 `django.contrib.sites` *?* `Site` 模型*?* `domain` *?* `name` 两个字段
- `SITE_ID` 设置指定了与特定配置文件相关联的 `Site` 对象之数据库 *ID*

如何运用这两个概念由你决定，*?Django* 是通过几个简单的约定自动使用的

安装多站点应用要执行以下几个步骤

1. *?* `'django.contrib.sites'` 加入*?* `INSTALLED_APPS` 中
2. 运行 *<cite>manage.py syncdb</cite>* 命令*?<cite>django_site</cite>* 表安装到数据库中*?*这样也会建立默认的站点对象，域名*?example.com*
3. *?tt class="docutils literal">example.com*改成你自己的域名，然后通过*Django admin*站点或*Python API*来添加其*?tt class="docutils literal">Site*对象*?*为该 *Django* 项目支撑的每个站（或域）创建一*?* `Site` 对象
4. 在每个设置文件中定义一*?* `SITE_ID` 变量*?*该变量值应当是该设置文件所支撑的站*?tt class="docutils literal">Site* 对象的数据库 *ID*

多站点框架的功能

下面几节讲述的是用多站点框架能够完成的几项工作

多个站点的数据重

正如在情景一中所解释的，要在多个站点间重用数?仅需在模型中? `Site` 添加一? `多对多字` 即可，例如：

```
_from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(max_length=200)
    # ...
    sites = models.ManyToManyField(Site)_
```

这是在数据库中为多个站点进行文章关联操作的基础步骤?在适当的位置使用该技术，你可以在多个站点中重复使用同一?*Django* 视图代码?继续 `Article` 模型范例，下面是一个可能的 `article_detail` 视图

```
_from django.conf import settings
from django.shortcuts import get_object_or_404
from mysite.articles.models import Article

def article_detail(request, article_id):
    a = get_object_or_404(Article, id=article_id, sites__id=settings.SITE_ID)
    # ..._
```

该视图方法是可重用的，因为它根据 `SITE_ID` 设置的值动态检?*articles* 站点

例如?*LJWorld.coms* 设置文件中有有个 `SITE_ID` 设置? `1` ，?*Lawrence.coms* 设置文件中有? `SITE_ID` 设置? `2` 。如果该视图?*LJWorld.coms* 处于激活状态时被调用，那么它将把查找范围局限于站点列表包括 *LJWorld.com* 在内的文章

将内容与单一站点相关

同样，你也可以使? `外键` 在多对一关系中将一个模型关联到 `Site` 模型

举例来说，如果某篇文章仅仅能够出现在一个站点上，你可以使用下面这样的模型：

```
_from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(max_length=200)
    # ...
    site = models.ForeignKey(Site)_
```

这与前一节中介绍的一样有益

从视图钩挂当前站

在底层，通过?Django 视图中使用多站点框架，你可以让视图根据调用站点不同而完成不同的工作，例如：

```
_from django.conf import settings

def my_view(request):
    if settings.SITE_ID == 3:
        # Do something.
    else:
        # Do something else._
```

当然，像那样对站?ID 进行硬编码是比较难看的?略为简洁的完成方式是查看当前的站点域：

```
_from django.conf import settings
from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get(id=settings.SITE_ID)
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else._
```

? `Site` 对象中获? `settings.SITE_ID` 值的做法比较常见，因? `Site` 模型管理?(`Site.objects`) 具备一? `get_current()` 方法?下面的例子与前一个是等效的：

```
_from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get_current()
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else._
```

注意

在这个最后的例子里，你不用导? `django.conf.settings`

获取当前域用于呈

正如情景二中所解释的那样，依据*DRY*原则*(*不做重复工作*)*，你只需在一个位置储存站名和域名，然后引用当? `Site` 对象? `name` ? `domain` 。例如： 例如

```
_from django.contrib.sites.models import Site
from django.core.mail import send_mail

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...
    current_site = Site.objects.get_current()
    send_mail('Thanks for subscribing to %s alerts' % current_site.name,
        'Thanks for your subscription. We appreciate it.\n\n-The %s team.' % current_site
        'editor@%s' % current_site.domain,
        [user_email])
    # ..._
```

继续我们正在讨论?*LJWorld.com* ?*Lawrence.com* 例子，在*Lawrence.com* 该邮件的标题行是"感谢注?*Lawrence.com* 提醒信件"??*LJWorld.com* ，该邮件标题行是"感谢注?*LJWorld.com* 提醒信件"?这种站点关联行为方式对邮件信息主体也同样适用

完成这项工作的一种更加灵活（但更重量级）的方法是使用 *Django* 的模板系统?假定 *Lawrence.com* ?*LJWorld.com* 各自拥有不同的模板目录（ `TEMPLATE_DIRS` ），你可将工作轻松地转交给模板系统，如下所示：

```
_from django.core.mail import send_mail
from django.template import loader, Context

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...
    subject = loader.get_template('alerts/subject.txt').render(Context({}))
    message = loader.get_template('alerts/message.txt').render(Context({}))
    send_mail(subject, message, 'do-not-reply@example.com', [user_email])
    # ..._
```

本例中，你不得不?*LJWorld.com* ?*Lawrence.com* 的模板目录中都创建一? `subject.txt` ? `message.txt` 模板?正如之前所说，该方法带来了更大的灵活性，但也带来了更多复杂性

尽可能多的利? `Site` 对象是减少不必要的复杂、冗余工作的好办法

当前站点管理

如果 `站点` 在你的应用中扮演很重要的角色，请考虑在你的模型中使用方便? `CurrentSiteManager` ?这是一个模型管理器（见第十章），它会自动过滤使其只包含与当前站点相关联的对象

通过显示地将 `CurrentSiteManager` 加入模型中以使用它?例如

```
_from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(max_length=100)
    pub_date = models.DateField()
    site = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager()_
```

通过该模型， `Photo.objects.all()` 将返回数据库中所有的 `Photo` 对象，? `Photo.on_site.all()` 仅根? `SITE_ID` 设置返回与当前站点相关联? `Photo` 对象

换言之，以下两条语句是等效的

```
_Photo.objects.filter(site=settings.SITE_ID)
Photo.on_site.all()_
```

`CurrentSiteManager` 是如何知? `Photo` 的哪个字段是 `Site` 呢？缺省情况下，它会查找一个叫? `site` 的字段。如果你的模型包含了名字 `不是``site` ?em>外键或?tt class="docutils literal">多对关联，你需要把它作为参数传?tt class="docutils literal">CurrentSiteManager以显示指明。下面的模型拥有一?tt class="docutils literal">publish_on字段

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(max_length=100)
    pub_date = models.DateField()
    publish_on = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager('publish_on')
```

如果试图使用 `CurrentSiteManager` 并传入一个不存在的字段名?Django 将引发一? `ValueError` 异常

注意

即便是已经使用了 `CurrentSiteManager` ，你也许还想在模型中拥有一个正常的（非站点相关）的 `管理` 。正如在附录 B 中所解释的，如果你手动定义了一个管理器，那?Django 不会为你创建全自动的 `objects = models.Manager()` 管理器

同样，Django 的特定部分（?Django 超级管理站点和通用视图）使用在模型中定??em>第一管理器，因此如果希望管理站点能够访问所有对象（而不是仅仅站点特有对象），请于定? `CurrentSiteManager` 之前在模型中放入 `objects = models.Manager()`

Django如何使用多站点框

尽管并不是必须的，我们还是强烈建议使用多站点框架，因?Django 在几个地方利用了它?即使只用 Django 来支持单个网站，你也应该花一点时间用 `domain` ? `name` 来创建站点对象，并将 `SITE_ID` 设置指向它的 ID

以下讲述的是 Django 如何使用多站点框架：

- 在重定向框架中（见后面的重定向一节），每一个重定向对象都与一个特定站点关联?? Django 搜索重定向的时候，它会考虑当前? `SITE_ID`
- 在注册框架中，每个注释都与特定站点相关?每个注释被显示时，其 `site` 被设置为当前? `SITE_ID` ，而当通过适当的模板标签列出注释时，只有当前站点的注释将会显示
- ?flatpages 框架?(参见后面?Flatpages 一节），每?flatpage 都与特定的站点相关联?创建 flatpage 时，你都将指定它? `site` ，?flatpage 中间件在获取 flatpage 以显示它的过程中，将查看当前? `SITE_ID`
- ?syndication 框架中（参阅?13 章）? `title` ? `description` 的模板会自动访问变量 `{{ site }}` ，它其实是代表当前站点的 `Site` 对象?而且，如果你不指定一个合格的domain的话，提供目录URL的钩子将会使用当前“Site”对象的domain
- 在权限框架中（参见十四章），视图 `django.contrib.auth.views.login` 把当?tt class="docutils literal">Site名字和对象分别以 `{{ site_name }}` ?tt class="docutils literal">{{ site }}的形式传给了模板

Flatpages(简单页?

尽管通常情况下总是搭建运行数据库驱动的 Web 应用，有时你还是需要添加一两张一次性的静态页面，例如“关于”页面，或者“隐私策略”页面等等?可以用像 Apache 这样的标准Web服务器来处理这些静态页面，但却会给应用带来一些额外的复杂性，因为你必须操心怎么配置 Apache，还要设置权限让整个团队可以修改编辑这些文件，而且你还不能使用 Django 模板系统来统一这些页面的风格

这个问题的解决方案是使用位于 `django.contrib.flatpages` 开发包中的 Django 简单页面（flatpages）应用程序。该应用让你能够通过 Django 管理站点来管理这些一次性的页面，还可以让你使用 Django 模板系统指定它们使用哪个模板?它在后台使用Django模型，这意味着它把页面项别的数据一样保存在数据库中，也就是说你可以使用标准Django数据库API来存取页面

简单页面以它们?URL 和站点为键值?当创建简单页面时，你指定它与哪个URL以及和哪个站点相关联？（有关站点的更多信息，请查阅”多站点“一节。）

使用简单页

安装简单页面应用程序必须按照下面的步骤

1. 添加 `'django.contrib.flatpages'` ? `INSTALLED_APPS` 设置?tt class="docutils literal">django.contrib.flatpages依赖 `django.contrib.sites` ，所以确保它们都?tt class="docutils literal">INSTALLED_APPS里

2. ? `'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware'` 添加? `MIDDLEWARE_CLASSES` 设置中

3. 运行 `manage.py syncdb` 命令在数据库中创建必需的两个表

简单页面应用程序在数据库中创建两个表： `django_flatpage` ? `django_flatpage_sites` ? `django_flatpage` 只是?URL 映射到标题和一段文本内容? `django_flatpage_sites` 是一个多对多表，用于关联某个简单页面以及一个或多个站点

该应用捆绑的 `FlatPage` 模型? `django/contrib/flatpages/models.py` 进行定义，如下所示：

```
from django.db import models
from django.contrib.sites.models import Site

class FlatPage(models.Model):
    url = models.CharField(max_length=100, db_index=True)
    title = models.CharField(max_length=200)
    content = models.TextField(blank=True)
    enable_comments = models.BooleanField()
    template_name = models.CharField(max_length=70, blank=True)
    registration_required = models.BooleanField()
    sites = models.ManyToManyField(Site)
```

让我们逐项看看这些字段的含义：

- `url` ：该简单页面所处的 URL，不包括域名，但是包含前导斜?(例如 `/about/contact/`)
- `title` ：简单页面的标题?框架不对它作任何特殊处理?由你通过模板来显示它
- `content` ：简单页面的内容 (?HTML 页面)?框架不对它作任何特殊处理?由你负责使用模板来显示
- `enable_comments` ：是否允许该简单页面使用评论?框架不对它作任何特殊处理?你可在模板中检查该值并根据需要显示评论窗体
- `template_name` ：用来解析该简单页面的模板名称?这是一个可选项；如果未指定模板或该模板不存在，系统会退而使用默认模? `flatpages/default.html`
- `registration_required` ：是否注册用户才能查看此简单页面?该设置项集成?Djangos 验证/用户框架，该框架于第十四章详述
- `sites` ：该简单页面放置的站点?该项设置集成?Django 多站点框架，该框架在本章的“多站点”一节中有所阐述

你可以通过 Django 超级管理界面或?Django 数据?API 来创建简单页面?要了解更多内容，请查阅“添加、修改和删除简单页面”一节

一旦简单页面创建完成， `FlatpageFallbackMiddleware` 将完成（剩下）所有的工作?每当 Django 引发 404 错误，作为最后的办法，该中间件将根据所请求?URL 检查简单页面数据库?确切地说，它将使用所指定?URL以及 `SITE_ID` 设置对应的站?ID 查找一个简单页面

如果找到一个匹配项，它将载入该简单页面的模板（如果没有指定的话，将使用默认模? `flatpages/default.html` ）?同时，它把一个简单的上下文变?tt class="docutils literal">flatpage（一个简单页面对象）传递给模板?模板解析过程中，它实际用的是 `RequestContext`

如果 `FlatpageFallbackMiddleware` 没有找到匹配项，该请求继续如常处理

注意

该中间件仅在发生 404 （页面未找到）错误时被激活，而不会在 500 （服务器错误）或其他错误响应时被激活?还要注意的是必须考虑 `MIDDLEWARE_CLASSES` 的顺序问题?通常，你可以? `FlatpageFallbackMiddleware` 放在列表最后，因为它是最后的办法

添加、修改和删除简单页

可以用两种方式增加、变更或删除简单页面：

通过超级管理界面

如果已经激活了自动?Django 超级管理界面，你将会在超级管理页面的首页看到有个 Flatpages 区域?你可以像编辑系统中其它对象那样编辑简单页面

通过 **Python API**

前面已经提到，简单页面表现为 `django/contrib/flatpages/models.py` 中的标准 Django 模型。这样，你就可以使用Django数据库API来存取简单页面对象，例如

```
>>> from django.contrib.flatpages.models import FlatPage
>>> from django.contrib.sites.models import Site
>>> fp = FlatPage.objects.create(
...     url='/about/',
...     title='About',
...     content='<p>About this site...</p>',
...     enable_comments=False,
...     template_name='',
...     registration_required=False,
... )
>>> fp.sites.add(Site.objects.get(id=1))
>>> FlatPage.objects.get(url='/about/')
<FlatPage: /about/ -- About>
```

使用简单页面模

缺省情况下，系统使用模板 `flatpages/default.html` 来解析简单页面，但你也可以通过设定 `FlatPage` 对象? `template_name` 字段来更改特定简单页面的模板

你必须自己创? `flatpages/default.html` 模板?只需要在模板目录创建一? `flatpages` 目录，并? `default.html` 文件置于其中

简单页面模板只接受有一个上下文变量—? `flatpage` ，也就是该简单页面对象

以下是一? `flatpages/default.html` 模板范例:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
<head>
<title>{{ flatpage.title }}</title>
</head>
<body>
{{ flatpage.content|safe }}
</body>
</html>
```

注意我们使用?tt class="docutils literal">safe模板过滤器来允许 `flatpage.content` 引入原始HTML而不必转义

重定

通过将重定向存储在数据库中并将其视为 Django 模型对象，Django 重定向框架让你能够轻松地管理它们?比如说，你可以通过重定向框架告诉Django，把任何指向 `/music/` 的请求重定向? `/sections/arts/music/` 。当你需要在站点中移动一些东西时，这项功能就派上用场了——网站开发者应该穷尽一切办法避免出现坏链接

使用重定向框

安装重定向应用程序必须遵循以下步骤：

1. ? `'django.contrib.redirects'` 添加? `INSTALLED_APPS` 设置中
2. ? `'django.contrib.redirects.middleware.RedirectFallbackMiddleware'` 添加? `MIDDLEWARE_CLASSES` 设置中
3. 运行 `manage.py syncdb` 命令将所需的表添加到数据库中

`manage.py syncdb` 在数据库中创建了一? `django_redirect` 表?这是一个简单的查询表，只有 `site_id` 、old_path和new_path三个字段

你可以通过 Django 超级管理界面或?Django 数据?API 来创建重定向?要了解更多信息，请参阅“增加、变更和删除重定向”一节

一旦创建了重定向， `RedirectFallbackMiddleware` 类将完成所有的工作?每当 Django 应用引发一?404 错误，作为终极手段，该中间件将为所请求?URL 在重定向数据库中进行查找?确切地说，它将使用给定的 `old_path` 以及 `SITE_ID` 设置对应的站?ID 查找重定向设置?（查阅

前面的“多站点”一节可了解关于 `SITE_ID` 和多站点框架的更多细节） 然后，它将执行以下两个步骤：

- 如果找到了匹配项，并? `new_path` 非空，它将重定向? `new_path`
- 如果找到了匹配项，但 `new_path` 为空，它将发送一?410 (Gone) HTTP 头信息以及一个空（无内容）响应
- 如果未找到匹配项，该请求将如常处理

该中间件仅为 404 错误激活，而不会为 500 错误或其他任何状态码的响应所激活

注意必须考虑 `MIDDLEWARE_CLASSES` 的顺序?通常，你可以? `RedirectFallbackMiddleware` 放置在列表的最后，因为它是一种终极手段

注意

如果同时使用重定向和简单页面回退中间件， 必须考虑先检查其中的哪一个（重定向或简单页面）?我们建议将简单页面放在重定向之前（因此将简单页面中间件放置在重定向中间件之前），但你可能有不同想法

增加、变更和删除重定

你可以两种方式增加、变更和删除重定向：

通过管理界面

如果已经激活了全自动的 Django 超级管理界面，你应该能够在超级管理首页看到重定向区域?可以像编辑系统中其它对象一样编辑重定向

同过Python API

重定向表现为 `django/contrib/redirects/models.py` 中的一个标?Django 模型。因此，你可以通过Django数据库API来存取重定向对象，例如：

```
>>> from django.contrib.redirects.models import Redirect
>>> from django.contrib.sites.models import Site
>>> red = Redirect.objects.create(
...     site=Site.objects.get(id=1),
...     old_path='/music/',
...     new_path='/sections/arts/music/',
... )
>>> Redirect.objects.get(old_path='/music/')
<Redirect: /music/ ---> /sections/arts/music/>
```

CSRF 防护

`django.contrib.csrf` 开发包能够防止遭受跨站请求伪造攻?(CSRF).

CSRF, 又叫会话跳转，是一种网站安全攻击技术?当某个恶意网站在用户未察觉的情况下将其从一个已经通过身份验证的站点诱骗至一个新?URL 时，这种攻击就发生了，因此它可以利用用户已经通过身份验证的状态?乍一看，要理解这种攻击技术比较困难，因此我们在本节将使用两个例子来说明

一个简单的 CSRF 例子

假定你已经登录到 `example.com` 的网页邮件账号。该网站有一个指?tt class="docutils literal">example.com/logout的注销按钮。就是说，注销其实就是访问 `example.com/logout`

通过在（恶意）网页上用隐藏一个指?URL `example.com/logout` ? `<iframe>` ,恶意网站可以强迫你访问该 URL 。因此，如果你登? `example.com` 的网页邮件账号之后，访问了带有指? `example.com/logout` ? `<iframe>` 的恶意站点，访问该恶意页面的动作将使你登? `example.com` ?Thus, if you're logged in to the `example.com` webmail account and visit the malicious page that has an `<iframe>` to `example.com/logout` , the act of visiting the malicious page will log you out from `example.com` .

很明显，登出一个邮件网站也不是什么严重的安全问题。但是同样的攻击可能针对任何相信用户的站点，比如在线银行和电子商务网站。这样的话可能在用户不知情的情况下就下订单付款了

稍微复杂一点的CSRF例子

在上一个例子中? `example.com` 应该负部分责任，因为它允许通过 HTTP `GET` 方法进行状态变更（即登入和登出）?如果对服务器的状态变更要求使?HTTP `POST` 方法，情况就好得多了?但是，即便是强制要求使用 `POST` 方法进行状态变更操作也易受?CSRF 攻击

假设 `example.com` 对登出功能进行了升级，登? `<form>` 按钮是通过一个指?URL `example.com/logout` ? `POST` 动作完成，同时在 `<form>` 中加入了以下隐藏的字段：

```
<input type="hidden" name="confirm" value="true">
```

这就确保了用简单的指向 `example.com/logout` ?tt class="docutils literal">POST 不会让用户登出；要让用户登出，用户必须通过 `POST` ? `example.com/logout` 发送请? 并且发送一个值为'true'的POST变量? `confirm`

尽管增加了额外的安全机制，这种设计仍然会遭到 CSRF 的攻击——恶意页面仅需一点点改进而已?攻击者可以针对你的站点设计整个表单，并将其藏身于一个不可见? `<iframe>` 中，然后使用 Javascript 自动提交该表单

防止 CSRF

那么，是否可以让站点免受这种攻击呢？第一步，首先确保所? `GET` 方法没有副作用?这样以来，如果某个恶意站点将你的页面包含? `<iframe>` ，它将不会产生负面效果

该技术没有考虑 `POST` 请求?第二步就是给所? `POST` 的form标签一个隐藏字段，它的值是保密的并根据用户进程?ID 生成?这样，从服务器端访问表单时，可以检查该保密的字段。不吻合时可以引发一个错误

这正?Django CSRF 防护层完成的工作，正如下面的小节所介绍的

使用**CSRF**中间

`django.contrib.csrf` 开发包只有一个模块： `middleware.py` 。该模块包含了一?Django 中间件类—? `CsrfMiddleware` ，该类实现了 CSRF 防护功能

在设置文件中? `'django.contrib.csrf.middleware.CsrfMiddleware'` 添加? `MIDDLEWARE_CLASSES` 设置中可激?CSRF 防护?该中间件必须? `SessionMiddleware` 之后 执行，因此在列表? `CsrfMiddleware` 必须出现? `SessionMiddleware` 之前 （因为响应中间件是自后向前执行的）?同时，它也必须在响应被压缩或解压之前对响应结果进行处理，因此 `CsrfMiddleware` 必须? `GZipMiddleware` 之后执行。一旦将它添加到 `MIDDLEWARE_CLASSES` 设置中，你就完成了工作?参见第十五章的“MIDDLEWARE_CLASSES顺序”小节以了解更多

如果感兴趣的话，下面? `CsrfMiddleware` 的工作模式?它完成以下两项工作：

1. 它修改当前处理的请求，向所有的 `POST` 表单增添一个隐藏的表单字段，使用名称是 `csrfmiddlewaretoken` ，值为当前会话 ID 加上一个密钥的散列值?如果未设置会?ID ，该中间件将 不会 修改响应结果，因此对于未使用会话的请求来说性能损失是可以忽略的

2. 对于所有含会话 cookie 集合的传? `POST` 请求，它将检查是否存? `csrfmiddlewaretoken` 及其是否正确?如果不是的话，用户将会收到一?403 `HTTP` 错误?403 错误页面的内容是检测到了跨域请求伪装?终止请求

该步骤确保只有源自你的站点的表单才能将数?POST 回来

该中间件特意只针?HTTP `POST` 请求（以及对应的 POST 表单）?如我们所解释的，永远不应该因为使用了 `GET` 请求而产生负面效应，你必须自己来确保这一点

未使用会?cookie ? `POST` 请求无法受到保护，但它们也不 需 受到保护，因为恶意网站可用任意方法来制造这种请求

为了避免转换*?HTML* 请求，中间件在编辑响应结果之前对它的 *`Content-Type`* 头标进行检查*?* 只有标记*? `text/html` ? `application/xml+xhtml`* 的页面才会被修改

CSRF中间件的局限

`CsrfMiddleware` 的运行需*?Django* 的会话框架*?*（参阅第 *14* 章了解更多关于会话的内容。）如果你使用了自定义会话或者身份验证框架手动管理会*?cookies*，该中间件将帮不上你的忙

如果你的应用程序以某种非常规的方法创?HTML 页面（例如：?*Javascript ?tt class="docutils literal">document.write*语句中发?*HTML* 片段），你可能会绕开了向表单添加隐藏字段的过滤器?在此情况下，表单提交永远无法成功?（这是因为在页面发送到客户端之前，`CsrfMiddleware` 使用正则表达式来添加 `csrfmiddlewaretoken` 字段到你的*HTML*中，而正则表达式不能处理不规范的*HTML*。）如果你怀疑出现了这样的问题。使用你浏览器的查看源代码功能以确定 `csrfmiddlewaretoken` 是否插入到了表单中

想了解更多关?*CSRF* 的信息和例子的话，可以访?*http://en.wikipedia.org/wiki/CSRF*

人性化数据

*?tt class="docutils literal">django.contrib.humanize*包含了一些是数据更人性化的模板过滤器?要激活这些过滤器，请*?tt class="docutils literal">'django.contrib.humanize'*加入到你*?tt class="docutils literal">INSTALLED_APPS*中。完成之后，向模版了加入 `{% load humanize %}` 就可以使用下面的过滤器了

apnumber

对于 *1 ?9* 的数字，该过滤器返回了数字的拼写形式?否则，它将返回数字?这遵循的是美联社风格

举例

- *1* 变成 *one*
- *2* 变成 *two*
- *10* 变成 *10*

你可以传入一个整数或者表示整数的字符串

intcomma

该过滤器将整数转换为每三个数字用一个逗号分隔的字符串

例子

- *4500* 变成 *4,500*
- *45000* 变成 *45,000*
- *450000* 变成 *450,000*
- *4500000* 变成 *4,500,000*

可以传入整数或者表示整数的字符串

intword

该过滤器将一个很大的整数转换成友好的文本表示方式?它对于超过一百万的数字最好用

例子

- *1000000 变成 1.0 million*
- *1200000 变成 1.2 million*
- *1200000000 变成 1.2 billion*

最大支持不超过一千的五次方（*1,000,000,000,000,000*）

可以传入整数或者表示整数的字符串

ordinal

该过滤器将整数转换为序数词的字符串形式

例子

- *1 变成 1st*
- *2 变成 2nd*
- *3 变成 3rd*
- *254变成254th*

可以传入整数或者表示整数的字符串

标记过滤

*?tt class="docutils literal">django.contrib.markup*包含了一些列*Django*模板过滤器，每一个都实现了一中通用的标记语言

- `textile` *: 实现?Textile (http://en.wikipedia.org/wiki/Textile*%28markup*language%29)*
- `markdown` *: 实现?Markdown (http://en.wikipedia.org/wiki/Markdown)*
- `restructuredtext` *: 实现?ReStructured Text (http://en.wikipedia.org/wiki/ReStructuredText)*

每种情形下，过滤器都期望字符串形式的格式化标记，并返回表示标记文本的字符串?例如*?tt class="docutils literal">textile*过滤器吧*Textile*格式的文本转换为*HTML*

```
_{% load markup %}
{{ object.content|textile }}_
```

_要激活这些过滤器，仅需? `'django.contrib.markup'` 添加? `INSTALLED_APPS` 设置中?一旦完成了该项工作，在模板中通过 `{% load markup %}` 就能使用这些过滤器?要想掌握更多信息的话，可阅读 `django/contrib/markup/templatetags/markup.py.` 内的源代码

下一

这些继承框架（*CSRF*、身份验证系统等等）通过提供 中间 来实现其奇妙的功能。中间件是在请求之前/后执行的可以修改请求和响应的代码，它扩展了框架?在下一章，我们将介绍Django的中间件并解释怎样写出自己的中间件

第十七章：中间件

在有些场合，需要对Django处理的每个request都执行某段代码。 这类代码可能是在view处理之前修改传入的request，或者记录日志信息以便于调试，等等。

这类功能可以用Django的中间件框架来实现，该框架由切入到Django的request/response处理过程中的钩子集合组成。 这个轻量级低层次的plug-in系统，能用于全面的修改Django的输入和输出。

每个中间件组件都用于某个特定的功能。 如果你是顺着这本书读下来的话，你应该已经多次见到“中间件”了

- 第12章中所有的session和user工具都籍由一小簇中间件实现(例如，由中间件设定view中可见的 `request.session` 和 `request.user`)。
- 第13章讨论的站点范围cache实际上也是由一个中间件实现，一旦该中间件发现与view相应的response已在缓存中，就不再调用对应的view函数。
- 第14章所介绍的 `flatpages` , `redirects` , 和 `csrf` 等应用也都是通过中间件组件来完成其魔法般的功能。

这一章将深入到中间件及其工作机制中，并阐述如何自行编写中间件。

什么是中间件

我们从一个简单的例子开始。

高流量的站点通常需要将Django部署在负载平衡proxy(参见第20章)之后。 这种方式将带来一些复杂性，其一就是每个request中的远程IP地址(`request.META["REMOTE_IP"]`)将指向该负载平衡proxy，而不是发起这个request的实际IP。 负载平衡proxy处理这个问题的方法在特殊的 `X-Forwarded-For` 中设置实际发起请求的IP。

因此，需要一个小小的中间件来确保运行在proxy之后的站点也能够在 `request.META["REMOTE_ADDR"]` 中得到正确的IP地址：

```
class SetRemoteAddrFromForwardedFor(object):
    def process_request(self, request):
        try:
            real_ip = request.META['HTTP_X_FORWARDED_FOR']
        except KeyError:
            pass
        else:
            # HTTP_X_FORWARDED_FOR can be a comma-separated list of IPs.
            # Take just the first one.
            real_ip = real_ip.split(",")[0]
            request.META['REMOTE_ADDR'] = real_ip
```

(Note: Although the HTTP header is called `X-Forwarded-For` , Django makes it available as `request.META['HTTP_X_FORWARDED_FOR']` . With the exception of `content-length` and `content-type` , any HTTP headers in the request are converted to `request.META` keys by converting all characters to uppercase, replacing any hyphens with underscores and adding an `HTTP_` prefix to the name.)

一旦安装了该中间件(参见下一节)，每个request中的 `X-Forwarded-For` 值都会被自动插入到 `request.META['REMOTE_ADDR']` 中。这样，Django应用就不需要关心自己是否位于负载平衡proxy之后；简单读取 `request.META['REMOTE_ADDR']` 的方式在是否有proxy的情形下都将正常工作。

实际上，为针对这个非常常见的情形，Django已将该中间件内置。 它位于 `django.middleware.http` 中, 下一节将给出这个中间件相关的更多细节。

安装中间件

如果按顺序阅读本书，应当已经看到涉及到中间件安装的多个示例,因为前面章节的许多例子都需要某些特定的中间件。 出于完整性考虑，下面介绍如何安装中间件。

要启用一个中间件，只需将其添加到配置模块的 `MIDDLEWARE_CLASSES` 元组中。 在 `MIDDLEWARE_CLASSES` 中，中间件组件用字符串表示： 指向中间件类名的完整Python路径。 例如，下面是 `django-admin.py startproject` 创建的缺省 `MIDDLEWARE_CLASSES` ：

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
)
```

Django项目的安装并不强制要求任何中间件，如果你愿意， `MIDDLEWARE_CLASSES` 可以为空。

这里中间件出现的顺序非常重要。 在request和view的处理阶段，Django按照 `MIDDLEWARE_CLASSES` 中出现的顺序来应用中间件，而在response和异常处理阶段，Django则按逆序来调用它们。 也就是说，Django将 `MIDDLEWARE_CLASSES` 视为view函数外层的顺序包装子： 在request阶段按顺序从上到下穿过，而在response则反过来。

中间件方法

现在，我们已经知道什么是中间件和怎么安装它，下面将介绍中间件类中可以定义的所有方法。

Initializer: init(self) init(self)「初始化]

在中间件类中， `__init__()` 方法用于执行系统范围的设置。

出于性能的考虑，每个已启用的中间件在每个服务器进程中只初始化 一 次。 也就是说 `__init__()` 仅在服务进程启动的时候调用，而在针对单个request处理时并不执行。

对一个middleware而言，定义 `__init__()` 方法的通常原因是检查自身的必要性。 如果 `__init__()` 抛出异常 `django.core.exceptions.MiddlewareNotUsed` ,则Django将从middleware栈中移出该middleware。 可以用这个机制来检查middleware依赖的软件是否存在、服务是否运行于调试模式、以及任何其它环境因素。

在中间件中定义 `__init__()` 方法时，除了标准的 `self` 参数之外，不应定义任何其它参数。

Request预处理函数: process_request(self, request) process_request(self, request)

这个方法的调用时机在Django接收到request之后，但仍未解析URL以确定应当运行的view之前。 Django向它传入相应的 `HttpRequest` 对象，以便在方法中修改。

`process_request()` 应当返回 `None` 或 `HttpResponse` 对象.

- 如果返回 `None` , Django将继续处理这个request,执行后续的中间件， 然后调用相应的view.
- 如果返回 `HttpResponse` 对象, Django 将不再执行 任何 其它的中间件(而无视其种类)以及相应的view。 Django将立即返回该 `HttpResponse` .

View预处理函数: process_view(self, request, view, args, kwargs) process_view(self, request, view, args, kwargs)

这个方法的调用时机在Django执行完request预处理函数并确定待执行的view之后，但在view函数实际执行之前。

表15-1列出了传入到这个View预处理函数的参数。

表 15-1\. 传入process_view()的参数

参数	说明
\`request\`	The \`HttpRequest\` object.
\`view\`	The Python function that Django will call to handle this request. This is the actual function object itself, not the name of the function as a string.
\`args\`	将传入view的位置参数列表，但不包括 \`request\` 参数(它通常是传 入view的第一个参数)
\`kwargs\`	将传入view的关键字参数字典.

Just like `process_request()` , `process_view()` should return either `None` or an `HttpResponse` object.

- If it returns `None` , Django will continue processing this request, executing any other middleware and then the appropriate view.
- If it returns an `HttpResponse` object, Django won't bother calling *any* other middleware (of any type) or the appropriate view. Django will immediately return that `HttpResponse` .

Response后处理函数: process_response(self, request, response) process_response(self, request, response)

这个方法的调用时机在Django执行view函数并生成response之后。 Here, the processor can modify the content of a response. One obvious use case is content compression, such as gzipping of the request's HTML.

这个方法的参数相当直观: `request` 是request对象，而 `response` 则是从view中返回的response对象。 `request` is the request object, and `response` is the response object returned from the view.

不同可能返回 `None` 的request和view预处理函数, `process_response()` 必须 返回 `HttpResponse` 对象. 这个response对象可以是传入函数的那一个原始对象(通常已被修改)，也可以是全新生成的。 That response could be the original one passed into the function (possibly modified) or a brand-new one.

Exception后处理函数: process_exception(self, request, exception) process_exception(self, request, exception)

这个方法只有在request处理过程中出了问题并且view函数抛出了一个未捕获的异常时才会被调用。 这个钩子可以用来发送错误通知，将现场相关信息输出到日志文件, 或者甚至尝试从错误中自动恢复。

这个函数的参数除了一贯的 `request` 对象之外，还包括view函数抛出的实际的异常对象 `exception` 。

`process_exception()` 应当返回 None 或 HttpResponse 对象.

- 如果返回 `None` , Django将用框架内置的异常处理机制继续处理相应request。
- 如果返回 `HttpResponse` 对象, Django 将使用该response对象，而短路框架内置的异常处理机制。

备注

Django自带了相当数量的中间件类(将在随后章节介绍)，它们都是相当好的范例。 阅读这些代码将使你对中间件的强大有一个很好的认识。

在Djangos wiki上也可以找到大量的社区贡献的中间件范例:
http://code.djangoproject.com/wiki/ContributedMiddleware
http://code.djangoproject.com/wiki/ContributedMiddleware

内置的中间件

Django自带若干内置中间件以处理常见问题，将从下一节开始讨论。

认证支持中间件

中间件类: `django.contrib.auth.middleware.AuthenticationMiddleware` .
`django.contrib.auth.middleware.AuthenticationMiddleware` .

这个中间件激活认证支持功能. 它在每个传入的 `HttpRequest` 对象中添加代表当前登录用户的 `request.user` 属性。 It adds the `request.user` attribute, representing the currently logged-in user, to every incoming `HttpRequest` object.

完整的细节请参见第12章。

通用中间件

Middleware class: `django.middleware.common.CommonMiddleware` .

这个中间件为完美主义者提供了一些便利:

> 禁止 `DISALLOWED_USER_AGENTS` 列表中所设置的*user agent*访问 ：一旦提供，这一列表应当由已编译的正则表达式对象组成，这些对象用于匹配传入的request请求头中的user-agent域。 下面这个例子来自某个配置文件片段：

```
import re

DISALLOWED_USER_AGENTS = (
    re.compile(r'^OmniExplorer_Bot'),
    re.compile(r'^Googlebot')
)
```

> 请注意 `import re` ,因为 `DISALLOWED_USER_AGENTS` 要求其值为已编译的正则表达式(也就是 `re.compile()` 的返回值)。
>
> 依据 *`APPEND_SLASH`* 和 *`PREPEND_WWW`* 的设置执行*URL*重写 ：如果 `APPEND_SLASH` 为 `True` , 那些尾部没有斜杠的URL将被重定向到添加了斜杠的相应URL，除非path的最末组成部分包含点号。 因此， `foo.com/bar` 会被重定向到 `foo.com/bar/` , 但是 `foo.com/bar/file.txt` 将以不变形式通过。
>
> 如果 `PREPEND_WWW` 为 True , 那些缺少先导www.的URLs将会被重定向到含有先导www.的相应URL上。 will be redirected to the same URL with a leading www..
>
> 这两个选项都是为了规范化URL。 其后的哲学是每个URL都应且只应当存在于一处。 技术上来说，URL `example.com/bar` 与 `example.com/bar/` 及 `www.example.com/bar/` 都互不相同。
>
> 依据 *`USE_ETAGS`* 的设置处理*Etag* : *ETags* 是HTTP级别上按条件缓存页面的优化机制。 如果 `USE_ETAGS` 为 `True` ，Django针对每个请求以MD5算法处理页面内容，从而得到Etag, 在此基础上，Django将在适当情形下处理并返回 `Not Modified` 回应(译注：
>
> 请注意，还有一个条件化的 `GET` 中间件, 处理Etags并干得更多，下面马上就会提及。

压缩中间件

中间件类 `django.middleware.gzip.GZipMiddleware` .

这个中间件自动为能处理gzip压缩(包括所有的现代浏览器)的浏览器自动压缩返回]内容。 这将极大地减少Web服务器所耗用的带宽。 代价是压缩页面需要一些额外的处理时间。

相对于带宽，人们一般更青睐于速度，但是如果你的情形正好相反，尽可启用这个中间件。

条件化的**GET**中间件

Middleware class: `django.middleware.http.ConditionalGetMiddleware` .

这个中间件对条件化 `GET` 操作提供支持。 如果response头中包括 `Last-Modified` 或 `ETag` 域，并且request头中包含 `If-None-Match` 或 `If-Modified-Since` 域，且两者一致，则该response将被response 304(Not modified)取代。 对 `ETag` 的支持依赖于 `USE_ETAGS` 配置及事先在response头中设置 `ETag` 域。稍前所讨论的通用中间件可用于设置response中的 `ETag` 域。 As discussed above, the `ETag` header is set by the Common middleware.

此外，它也将删除处理 `HEAD` request时所生成的response中的任何内容，并在所有request的response头中设置 `Date` 和 `Content-Length` 域。

反向代理支持 **(X-Forwarded-For**中间件**)**

Middleware class: `django.middleware.http.SetRemoteAddrFromForwardedFor` .

这是我们在 什么是中间件 这一节中所举的例子。 在 `request.META['HTTP_X_FORWARDED_FOR']` 存在的前提下，它根据其值来设置 `request.META['REMOTE_ADDR']` 。在站点位于某个反向代理之后的、每个request的 `REMOTE_ADDR` 都被指向 `127.0.0.1` 的情形下，这一功能将非常有用。 It sets `request.META['REMOTE_ADDR']` based on `request.META['HTTP_X_FORWARDED_FOR']` , if the latter is set. This is useful if you're sitting behind a reverse proxy that causes each request's `REMOTE_ADDR` to be set to `127.0.0.1` .

红色警告！

这个middleware并 不 验证 `HTTP_X_FORWARDED_FOR` 的合法性。

如果站点并不位于自动设置 `HTTP_X_FORWARDED_FOR` 的反向代理之后，请不要使用这个中间件。 否则，因为任何人都能够伪造 `HTTP_X_FORWARDED_FOR` 值，而 `REMOTE_ADDR` 又是依据 `HTTP_X_FORWARDED_FOR` 来设置，这就意味着任何人都能够伪造IP地址。

只有当能够绝对信任 `HTTP_X_FORWARDED_FOR` 值得时候才能够使用这个中间件。

会话支持中间件

Middleware class: `django.contrib.sessions.middleware.SessionMiddleware` .

这个中间件激活会话支持功能. 细节请参见第12章。 See Chapter 14 for details.

站点缓存中间件

Middleware classes: `django.middleware.cache.UpdateCacheMiddleware` and `django.middleware.cache.FetchFromCacheMiddleware` .

这些中间件互相配合以缓存每个基于Django的页面。 已在第13章中详细讨论。

事务处理中间件

Middleware class: `django.middleware.transaction.TransactionMiddleware` .

这个中间件将数据库的 `COMMIT` 或 `ROLLBACK` 绑定到request/response处理阶段。 如果view函数成功执行，则发出 `COMMIT` 指令。 如果view函数抛出异常，则发出 `ROLLBACK` 指令。

这个中间件在栈中的顺序非常重要。 其外层的中间件模块运行在Django缺省的 保存-提交 行为模式下。 而其内层中间件(在栈中的其后位置出现)将置于与view函数一致的事务机制的控制下。

关于数据库事务处理的更多信息，请参见附录C。

下一章

Web开发者和数据库模式设计人员并不总是享有白手起家打造项目的奢侈机会。 In the next chapter, we’ll cover how to integrate with legacy systems, such as database schemas you’ve inherited from the 1980s.

第十八章： 集成已有的数据库和应用

Django最适合于所谓的green-field开发，即从头开始的一个项目，正如你在一块还长着青草的未开垦的土地上从零开始建造一栋建筑一般。 然而，尽管Django偏爱从头开始的项目，将这个框架和以前遗留的数据库和应用相整合仍然是可能的。 本章就将介绍一些整合的技巧。

与遗留数据库整合

Django的数据库层从Python代码生成SQL schemas—但是对于遗留数据库，你已经拥有SQL schemas. 这种情况,你需要为已经存在的数据表创建model. 为此,Django自带了一个可以通过读取您的数据表结构来生成model的工具. 该辅助工具称为inspectdb,你可以通过执行 `manage.py inspectdb` 来调用它.

使用 `inspectdb`

`inspectdb` 工具自省你配置文件指向的数据库，针对每一个表生成一个Django模型，然后将这些Python模型的代码显示在系统的标准输出里面。

下面是一个从头开始的针对一个典型的遗留数据库的整合过程。 两个前提条件是安装了Django和一个传统数据库。

> 通过运行django-admin.py startproject mysite (这里 `mysite` 是你的项目的名字)建立一个Django项目。 好的，那我们在这个例子中就用这个 `mysite` 作为项目的名字。
>
> 编辑项目中的配置文件, `mysite/settings.py` ,告诉Django你的数据库连接参数和数据库名。 具体的说，要提供 `DATABASE_NAME` , `DATABASE_ENGINE` , `DATABASE_USER` , `DATABASE_PASSWORD` , `DATABASE_HOST` , 和 `DATABASE_PORT` 这些配置信息.。 (请注意其中的一些设置是可选的。 更多信息参见第5章)
>
> 通过运行 `python mysite/manage.py startapp myapp` (这里 `myapp` 是你的应用的名字)创建一个Django应用。 这里我们使用 `myapp` 做为应用名。
>
> 运行命令 `python mysite/manage.py inspectdb` 。这将检查 `DATABASE_NAME` 数据库中所有的表并打印出为每张表生成的模型类。 看一看输出结果以了解inspectdb能做些什么。
>
> 将标准shell的输出重定向，保存输出到你的应用的 `models.py` 文件里：

```
python mysite/manage.py inspectdb > mysite/myapp/models.py
```

> 编辑 `mysite/myapp/models.py` 文件以清理生成的 models 并且做一些必要的自定义。 针对这个，下一个节有些好的建议。

清理生成的Models

如你可能会预料到的，数据库自省不是完美的，你需要对产生的模型代码做些许清理。 这里提醒一点关于处理生成 models 的要点:

> 数据库的每一个表都会被转化为一个model类 (也就是说，数据库的表和model 类之间是一对一的映射)。 这意味着你需要为多对多连接的表，重构其models 为 `ManyToManyField` 的对象。
>
> 所生成的每一个model中的每个字段都拥有自己的属性，包括id主键字段。 但是，请注意，如果某个model没有主键的话，那么Django会自动为其增加一个id主键字段。 这样一来，你也许希望移除这样的代码行。

```
id = models.IntegerField(primary_key=True)
```

> 这样做并不是仅仅因为这些行是冗余的，而且如果当你的应用需要向这些表中增加新记录时，这些行会导致某些问题。
>
> 每一个字段类型，如CharField、DateField， 是通过查找数据库列类型如VARCHAR,DATE来确定的。如果inspectdb无法把某个数据库字段映射到model字段上，它会使用TextField字段进行代替，并且会在所生成model字段后面加入Python注释“该字段类型是猜的”。 对这要当心，如果必要的话，更改字段类型。
>
> 如果你的数据库中的某个字段在Django中找不到合适的对应物，你可以放心的略过它。Django模型层不要求必须导入你数据库表中的每个列。
>
> 如果数据库中某个列的名字是Python的保留字（比如pass、class或者for等），inspectdb会在每个属性名后附加上_field，并将db_column属性设置为真实的字段名（也就是pass,class或者for等）。
>
> 例如，某张表中包含一个INT类型的列，其列名为for，那么所生成的model将会包含如下所示的一个字段：

```
for_field = models.IntegerField(db_column='for')
```

> `inspectdb` 会在该字段后加注 `'字段重命名，因为它是一个Python保留字'` 。
>
> 如果数据库中某张表引用了其他表（正如大多数数据库系统所做的那样），你需要适当的修改所生成model的顺序，以使得这种引用能够正确映射。 例如，model Book拥有一个针对于model Author的外键，那么后者应该先于前者被定义。如果你想创建一个指向尚未定义的model的关系，那么可以使用包含model名的字符串，而不是model对象本身。
>
> 对于PostgreSQL,MySQL和SQLite数据库系统，inspectdb能够自动检测出主键关系。 也就是说，它会在合适的位置插入primary_key=True。 而对于其他数据库系统，你必须为每一个model中至少一个字段插入这样的语句，因为Django的model要求必须拥有一个primary_key=True的字段。
>
> 外键检测仅对PostgreSQL，还有MySQL表中的某些特定类型生效。 至于其他数据库，外键字段将在假定其为INT列的情况下被自动生成为IntegerField。

与认证系统的整合

将Django与其他现有认证系统的用户名和密码或者认证方法进行整合是可以办到的。

例如，你所在的公司也许已经安装了LDAP，并且为每一个员工都存储了相应的用户名和密码。 如果用户在LDAP和基于Django的应用上拥有独立的账号，那么这时无论对于网络管理员还是用户自己来说，都是一件很令人头痛的事儿。

为了解决这样的问题，Django认证系统能让您以插件方式与其他认证资源进行交互。 您可以覆盖Diango默认的基于数据库的模式，您还可以使用默认的系统与其他系统进行交互。

指定认证后台

在后台，Django维护了一个用于检查认证的后台列表。 当某个人调用 `django.contrib.auth.authenticate()` (如14章中所述)时，Django会尝试对其认证后台进行遍历认证。 如果第一个认证方法失败，Django会尝试认证第二个，以此类推，一直到尝试完。

认证后台列表在AUTHENTICATION_BACKENDS设置中进行指定。 它应该是指向知道如何认证的Python类的Python路径的名字数组。 这些类可以在你Python路径的任何位置。

默认情况下，AUTHENTICATION_BACKENDS被设置为如下：

```
('django.contrib.auth.backends.ModelBackend',)
```

那就是检测Django用户数据库的基本认证模式。

AUTHENTICATION_BACKENDS的顺序很重要，如果用户名和密码在多个后台中都是有效的，那么Django将会在第一个正确匹配后停止进一步的处理。

编写认证后台

一个认证后台其实就是一个实现了如下两个方法的类： `get_user(id)` 和 `authenticate(**credentials)` 。

方法 `get_user` 需要一个参数 `id` ，这个 `id` 可以是用户名，数据库ID或者其他任何数值，该方法会返回一个 `User` 对象。

方法 `authenticate` 使用证书作为关键参数。 大多数情况下，该方法看起来如下：

```
class MyBackend(object):
    def authenticate(self, username=None, password=None):
        # Check the username/password and return a User.
```

但是有时候它也可以认证某个短语，例如：

```
class MyBackend(object):
    def authenticate(self, token=None):
        # Check the token and return a User.
```

每一个方法中， `authenticate` 都应该检测它所获取的证书，并且当证书有效时，返回一个匹配于该证书的 `User` 对象，如果证书无效那么返回 `None` 。 如果它们不合法，就返回 `None` 。

如14章中所述，Django管理系统紧密连接于其自己后台数据库的 `User` 对象。 实现这个功能的最好办法就是为您的后台数据库（如LDAP目录，外部SQL数据库等）中的每个用户都创建一个对应的Django User对象。 您可以提前写一个脚本来完成这个工作，也可以在某个用户第一次登陆的时候在 `authenticate` 方法中进行实现。

以下是一个示例后台程序，该后台用于认证定义在 `setting.py` 文件中的username和password变量，并且在该用户第一次认证的时候创建一个相应的Django `User` 对象。

```
from django.conf import settings
from django.contrib.auth.models import User, check_password

class SettingsBackend(object):
    """
    Authenticate against the settings ADMIN_LOGIN and ADMIN_PASSWORD.

    Use the login name, and a hash of the password. For example:

    ADMIN_LOGIN = 'admin'
    ADMIN_PASSWORD = 'sha1$4e987$afbcf42e21bd417fb71db8c66b321e9fc33051de'
    """
    def authenticate(self, username=None, password=None):
        login_valid = (settings.ADMIN_LOGIN == username)
        pwd_valid = check_password(password, settings.ADMIN_PASSWORD)
        if login_valid and pwd_valid:
            try:
                user = User.objects.get(username=username)
            except User.DoesNotExist:
                # Create a new user. Note that we can set password
                # to anything, because it won't be checked; the password
                # from settings.py will.
                user = User(username=username, password='get from settings.py')
                user.is_staff = True
                user.is_superuser = True
                user.save()
            return user
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

更多认证模块的后台, 参考Django文档。

和遗留Web应用集成

同由其他技术驱动的应用一样，在相同的Web服务器上运行Django应用也是可行的。 最简单直接的办法就是利用Apaches配置文件httpd.conf，将不同的URL类型分发至不同的技术。 （请注意，第12章包含了在Apache/mod_python上配置Django的相关内容，因此在尝试本章集成之前花些时间去仔细阅读第12章或许是值得的。）

关键在于只有在您的httpd.conf文件中进行了相关定义，Django对某个特定的URL类型的驱动才会被激活。 在第12章中解释的缺省部署方案假定您需要Django去驱动某个特定域上的每一个页面。

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

这里, `<Location "/">` 这一行表示用Django处理每个以根开头的URL.

精妙之处在于Django将<location>指令值限定于一个特定的目录树上。 举个例子，比如说您有一个在某个域中驱动大多数页面的遗留PHP应用，并且您希望不中断PHP代码的运行而在../admin/位置安装一个Django域。 要做到这一点，您只需将<location>值设置为/admin/即可。

```
<Location "/admin/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
</Location>
```

有了这样的设置，只有那些以/admin/开头的URL地址才会触发Django去进行处理。 其他页面会使用已存在的设置。

请注意，把Diango绑定到的合格的URL（比如在本章例子中的 `/admin/` ）并不会影响其对URL的解析。 绝对路径对Django才是有效的（例如 `/admin/people/person/add/` ），而非截断后的URL（例如 `/people/person/add/` ）。这意味着你的根URLconf必须包含前缀 `/admin/` 。

下一章

如果你的母语是英语, 你可能就不会注意到许多Django admin网站中最酷的特性功能。 它支持超过50种语言! Django 的国际化框架使其成为可能(还有Django志愿翻译者的努力)。下一章介绍如何使用这个框架来提供本地化的Django网站。

第十九章： 国际化

Django诞生于美国中部堪萨斯的劳伦斯，距美国的地理中心不到40英里。 像大多数开源项目一样，Djano社区逐渐开始包括来自全球各地的许多参与者。 鉴于Django社区逐渐变的多样性，国际化和本地化逐渐变得很重要。 由于很多开发者对这些措辞比较困惑，所以我们将简明的定义一下它们。

- 国际化* 是指为了该软件在任何地区的潜在使用而进行程序设计的过程。 它包括了为将来翻译而标记的文本（比如用户界面要素和错误信息等）、日期和时间的抽象显示以便保证不同地区的标准得到遵循、为不同时区提供支持，并且一般确保代码中不会存在关于使用者所在地区的假设。 您会经常看到国际化被缩写为“I18N”(18表示Internationlization这个单词首字母I和结尾字母N之间的字母有18个)。
- 本地化* 是指使一个国际化的程序为了在某个特定地区使用而进行实际翻译的过程。 有时，本地化缩写为 *L10N* 。

Django本身是完全国际化了的，所有的字符串均因翻译所需而被标记，并且设定了与地域无关的显示控制值，如时间和日期。 Django是带着50个不同的本地化文件发行的。 即使您的母语不是英语，Django也很有可能已经被翻译为您的母语了。

这些本地化文件所使用的国际化框架同样也可以被用在您自己的代码和模板中。

您只需要添加少量的挂接代码到您的Python代码和模板中。 这些挂接代码被称为 翻译字符串 。它们告诉Django：如果这段文本的译文可用的话，它应被翻译为终端用户指定的语言。

Django会根据用户的语言偏好，在线地运用这些挂接指令去翻译Web应用程序。

本质上来说，Django做两件事情：

- 它让开发者和模板的作者指定他们的应用程序的哪些部分应该被翻译。
- Django根据用户的语言偏好来翻译Web应用程序。

备注:

Django的翻译机制是使用 GNU `gettext` (http://www.gnu.org/software/gettext/)，具体为Python自带的标准模块 `gettext` 。

如果您不需要国际化:

Django的国际化挂接是默认开启的，这可能会给Django的运行增加一点点开销。 如果您不需要国际化支持，那么您可以在您的设置文件中设置 `USE_I18N = False` 。 如果 `USE_I18N` 被设为 False ，那么Django会进行一些优化，而不加载国际化支持机制。

您也可以从您的 `TEMPLATE_CONTEXT_PROCESSORS` 设置中移除 `'django.core.context_processors.i18n'` 。

对你的Django应用进行国际化的三个步骤:

1. 第一步：在你的Python代码和模板中嵌入待翻译的字符串。
2. 第二步：把那些字符串翻译成你要支持的语言。
3. 第三步：在你的Django settings文件中激活本地中间件。

我们将详细地对以上步骤逐一进行描述。

1、 如何指定待翻译字符串

翻译字符串指定这段需要被翻译的文本。 这些字符串可以出现在您的Python代码和模板中。 而标记出这些翻译字符串则是您的责任；系统仅能翻译出它所知道的东西。

在Python 代码中

标准翻译

使用函数 `ugettext()` 来指定一个翻译字符串。 作为惯例，使用短别名 `_` 来引入这个函数以节省键入时间.

在下面这个例子中，文本 `"Welcome to my site"` 被标记为待翻译字符串：

```
from django.utils.translation import ugettext as _

def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

显然，你也可以不使用别名来编码。 下面这个例子和前面两个例子相同：

```
from django.utils.translation import ugettext

def my_view(request):
    output = ugettext("Welcome to my site.")
    return HttpResponse(output)
```

翻译字符串对于计算出来的值同样有效。 下面这个例子等同前面一种：

```
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

翻译对变量也同样有效。 这里是一个同样的例子：

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

（以上两个例子中，对于使用变量或计算值，需要注意的一点是Django的待翻译字符串检测工具， `make-messages.py` ，将不能找到这些字符串。 稍后，在 `makemessages` 中会有更多讨论。）

你传递给 `_()` 或 `gettext()` 的字符串可以接受占位符，由Python标准命名字符串插入句法指定的。 例如：

```
def my_view(request, m, d):
    output = _('Today is %(month)s %(day)s.') % {'month': m, 'day': d}
    return HttpResponse(output)
```

这项技术使得特定语言的译文可以对这段文本进行重新排序。 比如，一段英语译文可能是 `"Today is November 26."` ，而一段西班牙语译文会是 `"Hoy es 26 de Noviembre."` 使用占位符（月份和日期）交换它们的位置。

由于这个原因，无论何时当你有多于一个单一参数时，你应当使用命名字符串插入（例如： `%(day)s` ）来替代位置插入（例如： `%s` or `%d` ）。 如果你使用位置插入的话，翻译动作将不能重新排序占位符文本。

标记字符串为不操作

使用 `django.utils.translation.gettext_noop()` 函数来标记一个不需要立即翻译的字符串。 这个串会稍后从变量翻译。

使用这种方法的环境是，有字符串必须以原始语言的形式存储（如储存在数据库中的字符串）而在最后需要被翻译出来（如显示给用户时）。

惰性翻译

使用 `django.utils.translation.gettext_lazy()` 函数，使得其中的值只有在访问时才会被翻译，而不是在 `gettext_lazy()` 被调用时翻译。

例如：要翻译一个模型的 `help_text` ，按以下进行：

```
from django.utils.translation import ugettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=ugettext_lazy('This is the help text'))
```

在这个例子中， `ugettext_lazy()` 将字符串作为惰性参照存储，而不是实际翻译。 翻译工作将在字符串在字符串上下文中被用到时进行，比如在Django管理页面提交模板时。

在Python中，无论何处你要使用一个unicode 字符串（一个 `unicode` 类型的对象），您都可以使用一个 `ugettext_lazy()` 调用的结果。 一个 `ugettext_lazy()` 对象并不知道如何把它自己转换成一个字节串。如果你尝试在一个需要字节串的地方使用它，事情将不会如你期待的那样。 同样，你也不能在一个字节串中使用一个 unicode 字符串。所以，这同常规的Python行为是一致的。 例如：

```
# This is fine: putting a unicode proxy into a unicode string.
u"Hello %s" % ugettext_lazy("people")

# This will not work, since you cannot insert a unicode object
# into a bytestring (nor can you insert our unicode proxy there)
"Hello %s" % ugettext_lazy("people")
```

如果你曾经见到到像 `"hello"` 这样的输出，你就可能在一个字节串中插入了 `ugettext_lazy()` 的结果。 在您的代码中，那是一个漏洞。

如果觉得 `gettext_lazy` 太过冗长，可以用 `_` （下划线）作为别名，就像这样：

```
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

在Django模型中总是无一例外的使用惰性翻译。 为了翻译，字段名和表名应该被标记。（否则的话，在管理界面中它们将不会被翻译） 这意味着在 `Meta` 类中显式地编写 `verbose_nane` 和 `verbose_name_plural` 选项，而不是依赖于Django默认的 `verbose_name` 和 `verbose_name_plural` （通过检查model的类名得到）。

```
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=_('This is the help text'))
    class Meta:
        verbose_name = _('my thing')
        verbose_name_plural = _('mythings')
```

复数的处理

使用 `django.utils.translation.ungettext()` 来指定以复数形式表示的消息。 例如：

```
from django.utils.translation import ungettext

def hello_world(request, count):
    page = ungettext('there is %(count)d object',
        'there are %(count)d objects', count) % {
            'count': count,
        }
    return HttpResponse(page)
```

`ngettext` 函数包括三个参数： 单数形式的翻译字符串，复数形式的翻译字符串，和对象的个数（将以 `count` 变量传递给需要翻译的语言）。

模板代码

Django模板使用两种模板标签，且语法格式与Python代码有些许不同。 为了使得模板访问到标签，需要将 `{% load i18n %}` 放在模板最前面。

这个 `{% trans %}` 模板标记翻译一个常量字符串 (括以单或双引号) 或 可变内容：

```
<title>{% trans "This is the title." %}</title>
<title>{% trans myvar %}</title>
```

如果有 `noop` 选项，变量查询还是有效但翻译会跳过。 当空缺内容要求将来再翻译时，这很有用。

```
<title>{% trans "myvar" noop %}</title>
```

在一个带 `{% trans %}` 的字符串中，混进一个模板变量是不可能的。如果你的译文要求字符串带有变量(占位符placeholders)，请使用 `{% blocktrans %}` ：

```
{% blocktrans %}This string will have {{ value }} inside.{% endblocktrans %}
```

使用模板过滤器来翻译一个模板表达式，需要在翻译的这段文本中将表达式绑定到一个本地变量中：

```
{% blocktrans with value|filter as myvar %}
This will have {{ myvar }} inside.
{% endblocktrans %}
```

如果需要在 `blocktrans` 标签内绑定多个表达式，可以用 `and` 来分隔：

```
{% blocktrans with book|title as book_t and author|title as author_t %}
This is {{ book_t }} by {{ author_t }}
{% endblocktrans %}
```

为了表示单复数相关的内容，需要在 `{% blocktrans %}` 和 `{% endblocktrans %}` 之间使用 `{% plural %}` 标签来指定单复数形式，例如：

```
{% blocktrans count list|length as counter %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }} objects.
{% endblocktrans %}
```

其内在机制是，所有的块和内嵌翻译调用相应的 `gettext` 或 `ngettext` 。

每一个 `RequestContext` 可以访问三个指定翻译变量：

- `{{ LANGUAGES }}` 是一系列元组组成的列表，每个元组的第一个元素是语言代码，第二个元素是用该语言表示的语言名称。
- 作为一二字符串， `LANGUAGE_CODE` 是当前用户的优先语言。 例如： `en-us` 。（请参见下面的Django如何发现语言偏好）
- `LANGUAGE_BIDI` 就是当前地域的说明。 如果为真（True）， 它就是从右向左书写的语言，例如： 希伯来语，阿拉伯语。 如果为假（False）， 它就是从左到右书写的语言，如： 英语，法语，德语等。

如果你不用这个 `RequestContext` 扩展，你可以用3个标记到那些值：

```
{% get_current_language as LANGUAGE_CODE %}
{% get_available_languages as LANGUAGES %}
{% get_current_language_bidi as LANGUAGE_BIDI %}
```

这些标记亦要求一个 `{% load i18n %}` 。

翻译的hook在任何接受常量字符串的模板块标签内也是可以使用的。 此时，使用 `_()` 表达式来指定翻译字符串，例如：

```
{% some_special_tag _("Page not found") value|yesno:_("yes,no") %}
```

在这种情况下，标记和过滤器两个都会看到已经翻译的字符串，所有它们并不需要提防翻译操作。

备注:

在这个例子中，翻译结构将放过字符串 `"yes,no"` ，而不是单独的字符串 `"yes"` 和 `"no"` 。翻译的字符串将需要包括逗号以便过滤器解析代码明白如何分割参数。 例如， 一个德语翻译器可能会翻译字符串 `"yes,no"` 为 `"ja,nein"` (保持逗号原封不动)。

与惰性翻译对象一道工作

在模型和公用函数中，使用 `ugettext_lazy()` 和 `ungettext_lazy()` 来标记字符串是很普遍的操作。 当你在你的代码中其它地方使用这些对象时，你应当确定你不会意外地转换它们成一个字符串，因为它们应被尽量晚地转换（以便正确的地域生效） 这需要使用及个帮助函数。

拼接字符串： **string_concat()**

标准Python字符串拼接(`''.join([...])`) 将不会工作在包括惰性翻译对象的列表上。 作为替代，你可以使用 `django.utils.translation.string_concat()` ， 这个函数创建了一个惰性对象，其连接起它的内容 并且 仅当结果被包括在一个字符串中时转换它们为字符串 。 例如：

```
from django.utils.translation import string_concat
# ...
name = ugettext_lazy(u'John Lennon')
instrument = ugettext_lazy(u'guitar')
result = string_concat([name, ': ', instrument])
```

System Message: ERROR/3 (`<string>` , line 519)

Error in “cnid” directive: no content permitted.

```
.. cnid:: 109

  在这种情况下，当
```

System Message: WARNING/2 (`<string>` , line 523)

Explicit markup ends without a blank line; unexpected unindent.

`result` 自己被用与一个字符串时， `result` 中的惰性翻译将仅被转换为字符串(通常在模板渲染时间)。

allow_lazy() 修饰符

Django提供很多功能函数（如：取一个字符串作为他们的第一个参数并且对那个字符串做些什么）。(尤其在 `django.utils` 中) 这些函数被模板过滤器像在其他代码中一样直接使用。

如果你写你自己的类似函数并且与翻译打交道，当第一个参数是惰性翻译对象时，你会面临“做什么”的难题。 因为你可能在视图之外使用这个函数（并且因此当前线程的本地设置将会不正确），所以你不想立即转换其为一个字符串。

象这种情况，请使用 `django.utils.functional.allow_lazy()` 修饰符。 它修改这个函数以便假如第一个参数是一个惰性翻译， 这个函数的赋值会被延后直到它需要被转化为一个字符串为止。

例如：

```
from django.utils.functional import allow_lazy

def fancy_utility_function(s, ...):
    # Do some conversion on string 's'
    # ...
fancy_utility_function = allow_lazy(fancy_utility_function, unicode)
```

`allow_lazy()` 装饰符 采用了另外的函数来装饰，以及一定量的，原始函数可以返回的特定类型的额外参数 (`*args`)。 通常，在这里包括 `unicode` 就足够了并且确定你的函数将仅返回Unicode字符串。

使用这个修饰符意味着你能写你的函数并且假设输入是合适的字符串，然后在末尾添加对惰性翻译对象的支持。

2、如何创建语言文件

当你标记了翻译字符串，你就需要写出（或获取已有的）对应的语言翻译信息。 这里就是它如何工作的。

地域限制

Django不支持把你的应用本地化到一个连它自己都还没被翻译的地域。 在这种情况下，它将忽略你的翻译文件。 如果你想尝试这个并且Django支持它，你会不可避免地见到这样一个混合体—参杂着你的译文和来自Django自己的英文。 如果你的应用需要你支持一个Django中没有的地域，你将至少需要做一个Django core的最小翻译。

消息文件

第一步，就是为一种语言创建一个信息文件。 信息文件是包含了某一语言翻译字符串和对这些字符串的翻译的一个文本文件。 信息文件以 `.po` 为后缀名。

Django中带有一个工具， `bin/make-messages.py` ，它完成了这些文件的创建和维护工作。 运行以下命令来创建或更新一个信息文件：

```
django-admin.py makemessages -l de
```

其中 `de` 是所创建的信息文件的语言代码。 在这里，语言代码是以本地格式给出的。 例如，巴西地区的葡萄牙语为 `pt_BR` ，澳大利亚地区的德语为 `de_AT` 。

这段脚本应该在三处之一运行：

- Django项目根目录。
- 您Django应用的根目录。
- `django` 根目录（不是Subversion检出目录，而是通过 `$PYTHONPATH` 链接或位于该路径的某处）。 这仅和你为Django自己创建一个翻译时有关。

这段脚本遍历你的项目源树或你的应用程序源树并且提取出所有为翻译而被标记的字符串。它在 `locale/LANG/LC_MESSAGES` 目录下创建（或更新）了一个信息文件。针对上面的 `de` ，应该是 `locale/de/LC_MESSAGES/django.po` 。

作为默认， `django-admin.py makemessages` 检测每一个有 `.html` 扩展名的文件。 以备你要重载缺省值，使用 `--extension` 或 `-e` 选项指定文件扩展名来检测。

```
django-admin.py makemessages -l de -e txt
```

用逗号和（或）使用 `-e` 或 `--extension` 来分隔多项扩展名：

```
django-admin.py makemessages -l de -e html,txt -e xml
```

当创建JavaScript翻译目录时，你需要使用特殊的Django域：**not** `-e js` 。

没有gettext?

如果没有安装 `gettext` 组件， `make-messages.py` 将会创建空白文件。 这种情况下，安装 `gettext` 组件或只是复制英语信息文件(`conf/locale/en/LC_MESSAGES/django.po`)来作为一个起点；只是一个空白的翻译信息文件而已。

工作在Windows上么？

如果你正在使用Windows，且需要安装GNU gettext共用程序以便 `django-admin makemessages` 可以工作，请参看下面Windows小节中gettext部分以获得更多信息。

`.po` 文件格式很直观。 每个 `.po` 文件包含一小部分的元数据，比如翻译维护人员的联系信息，而文件的大部分内容是简单的翻译字符串和对应语言翻译结果的映射关系的列表。

举个例子，如果Django应用程序包括一个 `"Welcome to my site."` 的待翻译字符串 ，像这样：

```
_("Welcome to my site.")
```

则 `django-admin.py makemessages` 将创建一个 `.po` 文件来包含以下片段的消息：

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
msgstr ""
```

快速解释：

- `msgid` 是在源文件中出现的翻译字符串。 不要做改动。
- `msgstr` 是相应语言的翻译结果。 刚创建时它只是空字符串，此时就需要你来完成它。 注意不要丢掉语句前后的引号。
- 作为方便之处，每一个消息都包括：以 `#` 为前缀的一个注释行并且定位上边的 `msgid` 行，文件名和行号。

对于比较长的信息也有其处理方法。 `msgstr` （或 `msgid` ）后紧跟着的字符串为一个空字符串。 然后真正的内容在其下面的几行。 这些字符串会被直接连在一起。 同时，不要忘了字符串末尾的空格，因为它们会不加空格地连到一起。

若要对新创建的翻译字符串校验所有的源代码和模板，并且更新所有语言的信息文件，可以运行以下命令：

```
django-admin.py makemessages -a
```

编译信息文件

创建信息文件之后，每次对其做了修改，都需要将它重新编译成一种更有效率的形式，供 `gettext` 使用。可以使用 `django-admin.py compilemessages` 完成。

这个工具作用于所有有效的 `.po` 文件，创建优化过的二进制 `.mo` 文件供 `gettext` 使用。在你可以运行 `django-admin.py makemessages` 的目录下，运行 `django-admin.py compilemessages` ：

```
django-admin.py compilemessages
```

就是这样了。 你的翻译成果已经可以使用了。

Django如何处理语言偏好

一旦你准备好了翻译，如果希望在Django中使用，那么只需要激活这些翻译即可。

在这些功能背后，Django拥有一个灵活的模型来确定在安装和使用应用程序的过程中选择使用的语言。

要设定一个安装阶段的语种偏好，请设定 `LANGUAGE_CODE` 。如果其他翻译器没有找到一个译文，Django将使用这个语种作为缺省的翻译最终尝试。

如果你只是想要用本地语言来运行Django，并且该语言的语言文件存在，只需要简单地设置 `LANGUAGE_CODE` 即可。

如果要让每一个使用者各自指定语言偏好，就需要使用 `LocaleMiddleware` 。 `LocaleMiddleware` 使得Django基于请求的数据进行语言选择，从而为每一位用户定制内容。 它为每一个用户定制内容。

使用 `LocaleMiddleware` 需要在 `MIDDLEWARE_CLASSES` 设置中增加 `'django.middleware.locale.LocaleMiddleware'` 。 中间件的顺序是有影响的，最好按照依照以下要求：

- 保证它是第一批安装的中间件类。

- 因为 `LocalMiddleware` 要用到session数据，所以需要放在 `SessionMiddleware` 之后。
- 如果你使用 `CacheMiddleware` ,把 `LocaleMiddleware` 放在它后面。

例如， `MIDDLE_CLASSES` 可能会是如此：

```
MIDDLEWARE_CLASSES = (
   'django.contrib.sessions.middleware.SessionMiddleware',
   'django.middleware.locale.LocaleMiddleware',
   'django.middleware.common.CommonMiddleware',
)
```

（更多关于中间件的内容，请参阅第17章）

`LocaleMiddleware` 按照如下算法确定用户的语言:

- 首先，在当前用户的 session 的中查找 `django_language` 键；
- 如未找到，它会找寻一个cookie
- 还找不到的话，它会在 HTTP 请求头部里查找 `Accept-Language` ， 该头部是你的浏览器发送的，并且按优先顺序告诉服务器你的语言偏好。 Django会尝试头部中的每一个语种直到它发现一个可用的翻译。
- 以上都失败了的话, 就使用全局的 `LANGUAGE_CODE` 设定值。

备注：

> 在上述每一处，语种偏好应作为字符串，以标准的语种格式出现。 例如，巴西葡萄牙语是 `pt-br`
>
> 如果一个基本语种存在而亚语种没有指定，Django将使用基本语种。 比如，如果用户指定了 `de-at` （澳式德语）但Django只有针对 `de` 的翻译，那么 `de` 会被选用。
>
> 只有在 `LANGUAGES` 设置中列出的语言才能被选用。 若希望将语言限制为所提供语言中的某些（因为应用程序并不提供所有语言的表示），则将 `LANGUAGES` 设置为所希望提供语言的列表，例如： 例如：

```
LANGUAGES = (
  ('de', _('German')),
  ('en', _('English')),
)
```

> 上面这个例子限制了语言偏好只能是德语和英语（包括它们的子语言，如 `de-ch` 和 `en-us` ）。
>
> 如果自定义了 `LANGUAGES` ，将语言标记为翻译字符串是可以的，但是，请不要使用 `django.utils.translation` 中的 `gettext()` （决不要在settings文件中导入 `django.utils.translation` ，因为这个模块本身是依赖于settings，这样做会导致无限循环），而是使用一个“虚构的” `gettext()` 。
>
> 解决方案就是使用一个“虚假的” `gettext()` 。以 下是一个settings文件的例子：

```
ugettext = lambda s: s

LANGUAGES = (
    ('de', ugettext('German')),
    ('en', ugettext('English')),
)
```

> 这样做的话， `make-messages.py` 仍会寻找并标记出将要被翻译的这些字符串，但翻译不会在运行时进行，故而需要在任何使用 *LANGUAGES* 的代码中用“真实的” `ugettext()`。
>
> `LocaleMiddleware` 只能选择那些Django已经提供了基础翻译的语言。 如果想要在应用程序中对Django中还没有基础翻译的语言提供翻译，那么必须至少先提供该语言的基本的翻译。 例如，Django使用特定的信息ID来翻译日期和时间格式，故要让系统正常工作，至少要提供这些基本的翻译。
>
> 以英语的 `.po` 文件为基础，翻译其中的技术相关的信息，可能还包括一些使之生效的信息。
>
> 技术相关的信息ID很容易被认出来：它们都是大写的。 这些信息ID的翻译与其他信息不同:你需要提供其对应的本地化内容。 例如，对于 DATETIME_FORMAT （或 DATE_FORMAT 、 TIME_FORMAT ），应该提供希望在该语言中使用的格式化字符串。 格式被模板标签 `now` 用来识别格式字符串。

一旦 `LocaleMiddleware` 决定用户的偏好，它会让这个偏好作为 `request.LANGUAGE_CODE` 对每一个 `HttpRequest` 有效。请随意在你的视图代码中读一读这个值。 以下是一个简单的例子：

```
def hello_world(request):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to read Austrian German.")
    else:
        return HttpResponse("You prefer to read another language.")
```

注意，对于静态翻译（无中间件）而言，此语言在 `settings.LANGUAGE_CODE` 中，而对于动态翻译（中间件），它在 `request.LANGUAGE_CODE` 中。

在你自己的项目中使用翻译

Django使用以下算法寻找翻译：

- 首先，Django在该视图所在的应用程序文件夹中寻找 `locale` 目录。 若找到所选语言的翻译，则加载该翻译。
- 第二步，Django在项目目录中寻找 `locale` 目录。 若找到翻译，则加载该翻译。
- 最后，Django使用 `django/conf/locale` 目录中的基本翻译。

以这种方式，你可以创建包含独立翻译的应用程序，可以覆盖项目中的基本翻译。 或者，你可以创建一个包含几个应用程序的大项目，并将所有需要的翻译放在一个大的项目信息文件中。 决定权在你手中。

所有的信息文件库都是以同样方式组织的： 它们是：

- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PROJECTPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- 所有在settings文件中 `LOCALE_PATHS` 中列出的路径以其列出的顺序搜索 `<language>/LC_MESSAGES/django.(po|mo)`
- `$PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)`

要创建信息文件，也是使用 `django-admin.py makemessages.py` 工具，和Django信息文件一样。 需要做的就是进入正确的目录—— `conf/locale` （在源码树的情况下）或者 `locale/` （在应用程序信息或项目信息的情况下）所在的目录下。 同样地，使用 `compile-messages.py` 生成 `gettext` 需要使用的二进制 `django.mo` 文件。

您亦可运行 `django-admin.py compilemessages --settings=path.to.settings` 来使编译器处理所有存在于您 `LOCALE_PATHS` 设置中的目录。

应用程序信息文件稍微难以发现——因为它们需要 `LocaleMiddle` 。如果不使用中间件，Django只会处理Django的信息文件和项目的信息文件。

最后，需要考虑一下翻译文件的结构。 若应用程序要发放给其他用户，应用到其它项目中，可能需要使用应用程序相关的翻译。 但是，使用应用程序相关的翻译和项目翻译在使用 `make-messages` 时会产生古怪的问题。它会遍历当前路径下所有的文件夹，这样可能会把应用消息文件里存在的消息ID重复放入项目消息文件中。

最容易的解决方法就是将不属于项目的应用程序（因此附带着本身的翻译）存储在项目树之外。 这样做的话，项目级的 `make-messages` 将只会翻译与项目精确相关的，而不包括那些独立发布的应用程序中的字符串。

`set_language` 重定向视图

方便起见，Django自带了一个 `django.views.i18n.set_language` 视图，作用是设置用户语言偏好并重定向返回到前一页面。

在URLconf中加入下面这行代码来激活这个视图：

```
(r'^i18n/', include('django.conf.urls.i18n')),
```

（注意这个例子使得这个视图在 `/i18n/setlang/` 中有效。）

这个视图是通过 `GET` 方法调用的，在请求中包含了 `language` 参数。 如果session已启用，这个视图会将语言选择保存在用户的session中。 否则，它会以缺省名 `django_language` 在cookie中保存这个语言选择。(这个名字可以通过 `LANGUAGE_COOKIE_NAME` 设置来改变)

保存了语言选择后，Django根据以下算法来重定向页面：

- Django 在 `POST` 数据中寻找一个 `下一个` 参数。
- 如果 `next` 参数不存在或为空，Django尝试重定向页面为HTML头部信息中 `Referer` 的值。
- 如果 `Referer` 也是空的，即该用户的浏览器并不发送 `Referer` 头信息，则页面将重定向到 `/` （页面根目录）。

这是一个HTML模板代码的例子：

```
<form action="/i18n/setlang/" method="post">
<input name="next" type="hidden" value="/next/page/" />
<select name="language">
    {% for lang in LANGUAGES %}
    <option value="{{ lang.0 }}">{{ lang.1 }}</option>
    {% endfor %}
</select>
<input type="submit" value="Go" />
</form>
```

翻译与JavaScript

将翻译添加到JavaScript会引起一些问题：

- JavaScript代码无法访问一个 `gettext` 的实现。
- JavaScript 代码并不访问 .po或 .mo 文件；它们需要由服务器分发。
- 针对JavaScript的翻译目录应尽量小。

Django已经提供了一个集成解决方案： 它会将翻译传递给JavaScript，因此就可以在JavaScript中调用 `gettext` 之类的代码。

`javascript_catalog` 视图

这些问题的主要解决方案就是 `javascript_catalog` 视图。该视图生成一个JavaScript代码库，包括模仿 gettext 接口的函数，和翻译字符串的数组。 这些翻译字符串来自于你在info_dict或URI中指定的应用，工程或Django内核。

像这样使用：

```
js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns = patterns('',
    (r'^jsi18n/$', 'django.views.i18n.javascript_catalog', js_info_dict),
)
```

`packages` 里的每个字符串应该是Python中的点分割的包的表达式形式（和在 `INSTALLED_APPS` 中的字符串相同的格式），而且应指向包含 `locale` 目录的包。 如果指定了多个包，所有的目录会合并成一个目录。 如果有用到来自不同应用程序的字符串的JavaScript，这种机制会很有帮助。

你可以动态使用视图，将包放在urlpatterns里：

```
urlpatterns = patterns('',
    (r'^jsi18n/(?P<packages>\S+)/$', 'django.views.i18n.javascript_catalog'),
)
```

这样的话，就可以在URL中指定由加号（ `+` ）分隔包名的包了。 如果页面使用来自不同应用程序的代码，且经常改变，还不想将其放在一个大的目录文件中，对于这些情况，显然这是很有用的。 出于安全考虑，这些值只能是 `django.conf` 或 `INSTALLED_APPS` 设置中的包。

使用JavaScript翻译目录

要使用这个目录，只要这样引入动态生成的脚本：

```
<script type="text/javascript" src="/path/to/jsi18n/"></script>
```

这就是管理页面如何从服务器获取翻译目录。 当目录加载后，JavaScript代码就能通过标准的 `gettext` 接口进行访问：

```
document.write(gettext('this is to be translated'));
```

也有一个 `ngettext` 接口：

```
var object_cnt = 1 // or 0, or 2, or 3, ...
s = ngettext('literal for the singular case',
        'literal for the plural case', object_cnt);
```

甚至有一个字符串插入函数：

```
function interpolate(fmt, obj, named);
```

插入句法是从Python借用的，所以 `interpolate` 函数对位置和命名插入均提供支持：

> 位置插入 `obj` 包括一个JavaScript数组对象，元素值在它们对应于`fmt`的占位符中以它们出现的相同次序顺序插值 。 例如：

```
fmts = ngettext('There is %s object. Remaining: %s',
        'There are %s objects. Remaining: %s', 11);
s = interpolate(fmts, [11, 20]);
// s is 'There are 11 objects. Remaining: 20'
```

> 命名插入 通过传送为真（TRUE）的布尔参数`name`来选择这个模式。 `obj`包括一个JavaScript 对象或相关数组。 例如：

```
d = {
    count: 10
    total: 50
};

fmts = ngettext('Total: %(total)s, there is %(count)s object',
'there are %(count)s of a total of %(total)s objects', d.count);
s = interpolate(fmts, d, true);
```

但是，你不应重复编写字符串插值： 这还是JavaScript，所以这段代码不得不重复做正则表达式置换。 它不会和Python中的字符串插补一样快，因此只有真正需要的时候再使用它（例如，利用 `ngettext` 生成合适的复数形式）。

创建JavaScript翻译目录

你可以创建和更改翻译目录，就像其他

Django翻译目录一样，使用django-admin.py makemessages 工具。 唯一的差别是需要提供一个 `-d djangojs` 的参数，就像这样：

```
django-admin.py makemessages -d djangojs -l de
```

这样来创建或更新JavaScript的德语翻译目录。 和普通的Django翻译目录一样，更新了翻译目录后，运行 `compile-messages.py` 即可。

熟悉 `gettext` 用户的注意事项

如果你了解 `gettext` ，你可能会发现Django进行翻译时的一些特殊的东西：

- 字符串域为 `django` 或 `djangojs` 。字符串域是用来区别将数据存储在同一信息文件库（一般是 `/usr/share/locale/` ）的不同程序。django 域是为Python和模板翻译字符串服务的，被加载到全局翻译目录。 `djangojs` 域只是用来尽可能缩小JavaScript翻译的体积。
- Django不单独使用 `xgettext` ， 而是经过Python包装后的 `xgettext` 和msgfmt。这主要是为了方便。

Windows下的 `gettext`

对于那些要提取消息或编译消息文件的人们来说，需要的只有这么多。翻译工作本身仅仅包含编辑这个类型的现存文件，但如果你要创建你自己的消息文件，或想要测试或编译一个更改过的消息文件，你将需要这个 `gettext` 公用程序。

- > 从http://sourceforge.net/projects/gettext下载以下zip文件
    - `gettext-runtime-X.bin.woe32.zip`
    - `gettext-tools-X.bin.woe32.zip`
    - `libiconv-X.bin.woe32.zip`
- 在同一文件夹下展开这3个文件。（也就是 `C:\Program Files\gettext-utils` ）
- 更新系统路径：
    - `控制面板 > 系统> 高级 > 环境变量`
    - 在 `系统变量` 列表中，点击 `Path` ，点击 `Edit`
    - 把 `;C:\Program Files\gettext-utils\bin` 加到 `变量值` 字段的末尾。

只要 `xgettext --version` 命令正常工作，你亦可使用从别处获得的 `gettext` 的二进制代码。有些版本的0.14.4二进制代码被发现不支持这个命令。 不要试图与Django公用程序一起使用一个 `gettext` 。在一个windows命令提示窗口输入命令 `xgettext –version [`](#id15)将导致出现一个错误弹出窗口–“xgettext.exe产生错误并且将被windows关闭”。

System Message: WARNING/2 (`<string>` , line 1346); *backlink*

Inline literal start-string without end-string.

下一章

未章 将关注于安全，如何帮助你的用户和网站远离恶意软件的攻击。

第二十章：安全

Internet并不安全。

现如今，每天都会出现新的安全问题。 我们目睹过病毒飞速地蔓延，大量被控制的肉鸡作为武器来攻击其他人，与垃圾邮件的永无止境的军备竞赛，以及许许多多站点被黑的报告。

作为Web开发人员，我们有责任来对抗这些黑暗的力量。 每一个Web开发者都应该把安全看成是Web编程中的基础部分。 不幸的是，要实现安全是困难的。

Django试图减轻这种难度。 它被设计为自动帮你避免一些web开发新手（甚至是老手）经常会犯的错误。 尽管如此，需要弄清楚，Django如何保护我们，以及我们可以采取哪些重要的方法来使得我们的代码更加安全。

首先，一个重要的前提： 我们并不打算给出web安全的一个详尽的说明，因此我们也不会详细地解释每一个薄弱环节。 在这里，我们会给出Django所面临的安全问题的一个大概。

Web安全现状

如果你从这章中只学到了一件事情，那么它会是：

在任何条件下都不要相信浏览器端提交的数据。

你从不会知道HTTP连接的另一端会是谁。 可能是一个正常的用户，但是同样可能是一个寻找漏洞的邪恶的骇客。

从浏览器传过来的任何性质的数据，都需要近乎狂热地接受检查。 这包括用户数据（比如Web表单提交的内容）和带外数据（比如，HTTP头、cookies以及其他信息）。 要修改那些浏览器自动添加的元数据，是一件很容易的事。

在这一章所提到的所有的安全隐患都直接源自对传入数据的信任，并且在使用前不加处理。 你需要不断地问自己，这些数据从何而来。

SQL注入

SQL注入 是一个很常见的形式，在SQL注入中，攻击者改变web网页的参数（例如 `GET` / `POST` 数据或者URL地址），加入一些其他的SQL片段。 未加处理的网站会将这些信息在后台数据库直接运行。

这种危险通常在由用户输入构造SQL语句时产生。 例如，假设我们要写一个函数，用来从通信录搜索页面收集一系列的联系信息。 为防止垃圾邮件发送器阅读系统中的email，我们将在提供email地址以前，首先强制用户输入用户名。

```
def user_contacts(request):
    user = request.GET['username']
    sql = "SELECT * FROM user_contacts WHERE username = '%s';" % username
    # execute the SQL here...
```

备注

在这个例子中，以及在以下所有的“不要这样做”的例子里，我们都去除了大量的代码，避免这些函数可以正常工作。 我们可不想这些例子被拿出去使用。

尽管，一眼看上去，这一点都不危险，实际上却不尽然。

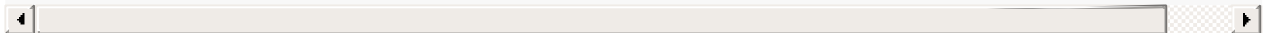
首先，我们对于保护email列表所采取的措施，遇到精心构造的查询语句就会失效。 想象一下，如果攻击者在查询框中输入 `"' OR 'a'='a"` 。 此时，查询的字符串会构造如下：

```
SELECT * FROM user_contacts WHERE username = '' OR 'a' = 'a';
```

由于我们允许不安全的SQL语句出现在字符串中，攻击者加入 `OR` 子句，使得每一行数据都被返回。

事实上，这是最温和的攻击方式。 如果攻击者提交了 `"'; DELETE FROM user_contacts WHERE 'a' = 'a'"` ，我们最终将得到这样的查询：

```
SELECT * FROM user_contacts WHERE username = ''; DELETE FROM user_contacts WHERE 'a' = 'a
```



哦！我们整个通信录名单去哪儿了？ 我们整个通讯录会被立即删除

解决方案

尽管这个问题很阴险，并且有时很难发现，解决方法却很简单： 绝不信任用户提交的数据，并且在传递给SQL语句时，总是转义它。

Django的数据库API帮你做了。 它会根据你所使用的数据库服务器（例如PostSQL或者MySQL）的转换规则，自动转义特殊的SQL参数。

举个例子，在下面这个API调用中：

```
foo.get_list(bar__exact="' OR 1=1")
```

Django会自动进行转义，得到如下表达：

```
SELECT * FROM foos WHERE bar = '\' OR 1=1'
```

完全无害。

这被运用到了整个Django的数据库API中，只有一些例外：

- 传给 `extra()` 方法的 `where` 参数。 (参考 附录 C。) 这个参数故意设计成可以接受原始的SQL。
- 使用底层数据库API的查询。 (详见第十章)

以上列举的每一个示例都能够很容易的让您的应用得到保护。 在每一个示例中，为了避免字符串被篡改而使用 绑定参数 来代替。这样，本节开始的例子应该写成这样：

```
from django.db import connection

def user_contacts(request):
    user = request.GET['username']
    sql = "SELECT * FROM user_contacts WHERE username = %s"
    cursor = connection.cursor()
    cursor.execute(sql, [user])
    # ... do something with the results
```

底层 `execute` 方法采用了一个SQL字符串作为其第二个参数，这个SQL字符串包含若干'%s'占位符，execute方法能够自动对传入列表中的参数进行转义和插入。 你应该用 *always* 这种方式构造自定义的SQL。

不幸的是，您并不是在SQL中能够处处都使用绑定参数，绑定参数不能够作为标识符（如表或列名等）。 因此，如果您需要这样做—我是说—动态构建 `POST` 变量中的数据库表的列表的话，您需要在您的代码中来对这些数据库表的名字进行转义。 Django提供了一个函数， `django.db.backend.quote_name` ，这个函数能够根据当前数据库引用结构对这些标识符进行转义。

跨站点脚本 (XSS)

在Web应用中， 跨站点脚本 (XSS)有时在被渲染成HTML之前，不能恰当地对用户提交的内容进行转义。 这使得攻击者能够向你的网站页面插入通常以 `<script>` 标签形式的任意HTML代码。

攻击者通常利用XSS攻击来窃取cookie和会话信息，或者诱骗用户将其私密信息透漏给被人（又称 钓鱼 ）。

这种类型的攻击能够采用多种不同的方式，并且拥有几乎无限的变体，因此我们还是只关注某个典型的例子吧。 让我们来想想这样一个极度简单的Hello World视图：

```
from django.http import HttpResponse

def say_hello(request):
    name = request.GET.get('name', 'world')
    return HttpResponse('<h1>Hello, %s!</h1>' % name)
```

这个视图只是简单的从GET参数中读取姓名然后将姓名传递给hello.html模板。 因此，如果我们访问 `http://example.com/hello/?name=Jacob` ，被呈现的页面将会包含一以下这些：

```
<h1>Hello, Jacob!</h1>
```

但是，等等，如果我们访问 `http://example.com/hello/?name=<i>Jacob</i>` 时又会发生什么呢？

```
<h1>Hello, <i>Jacob</i>!</h1>
```

当然，一个攻击者不会使用<i>标签开始的类似代码，他可能会用任意内容去包含一个完整的HTML集来劫持您的页面。 这种类型的攻击已经运用于虚假银行站点以诱骗用户输入个人信息，事实上这就是一种劫持XSS的形式，用以使用户向攻击者提供他们的银行帐户信息。

如果您将这些数据保存在数据库中，然后将其显示在您的站点上，那么问题就变得更严重了。 例如，一旦MySpace被发现这样的特点而能够轻易的被XSS攻击，后果不堪设想。 某个用户向他的简介中插入JavaScript，使得您在访问他的简介页面时自动将其加为您的好友，这样在几天之内，这个人就能拥有上百万的好友。 在几天的时间里，他拥有了数以百万的朋友。

现在，这种后果听起来还不那么恶劣，但是您要清楚——这个攻击者正设法将 他 的代码而不是MySpace的代码运行在 您 的计算机上。 这显然违背了假定信任——所有运行在MySpace上的代码应该都是MySpace编写的，而事实上却不如此。

MySpace是极度幸运的，因为这些恶意代码并没有自动删除访问者的帐户，没有修改他们的密码，也并没有使整个站点一团糟，或者出现其他因为这个弱点而导致的其他噩梦。

解决方案

解决方案是简单的： 总是转义可能来自某个用户的任何内容。

为了防止这种情况，Django的模板系统自动转义所有的变量值。 让我们来看看如果我们使用模板系统重写我们的例子会发生什么

```
# views.py

from django.shortcuts import render_to_response

def say_hello(request):
    name = request.GET.get('name', 'world')
    return render_to_response('hello.html', {'name': name})

# hello.html

<h1>Hello, {{ name }}!</h1>
```

这样，一个到 `http://example.com/hello/name=Jacob` 的请求将导致下面的页面：

```
<h1>Hello, &lt;i&gt;Jacob&lt;/i&gt;!</h1>
```

我们在第四章涵盖了Django的自动转义，一起想办法将其关闭。 甚至，如果Django真的新增了这些特性，您也应该习惯性的问自己，一直以来，这些数据都来自于哪里呢？ 没有哪个自动解决方案能够永远保护您的站点百分之百的不会受到XSS攻击。

伪造跨站点请求

伪造跨站点请求(CSRF)发生在当某个恶意Web站点诱骗用户不知不觉的从一个信任站点下载某个URL之时，这个信任站点已经被通过信任验证，因此恶意站点就利用了这个被信任状态。

Django拥有内建工具来防止这种攻击。 包括攻击本身及其使用的工具都在有详细介绍。16章

会话伪造/劫持

这不是某个特定的攻击，而是对用户会话数据的通用类攻击。 这种攻击可以采取多种形式：

> 中间人 攻击：检索所在有线（无线）网络，监听会话数据。
>
> 伪造会话 ：攻击者利用会话ID（可能是通过中间人攻击来获得）将自己伪装成另一个用户。
>
> 这两种攻击的一个例子可以是在一间咖啡店里的某个攻击者利用店内的无线网络来捕获某个会话cookie，然后她就可以利用那个cookie来假冒原始用户。 她便可以使该cookie来模拟原始用户。
>
> 伪造*cookie* ：就是指某个攻击者覆盖了在某个cookie中本应该是只读的数据。 `第十四章 <../chapter14/>` __ 详细介绍了cookies如何工作，以及要点之一的是，它在你不知道的情况下无视浏览器和恶意用户私自改变cookies。
>
> Web站点以 `IsLoggedIn=1` 或者 `LoggedInAsUser=jacob` 这样的方式来保存cookie由来已久，使用这样的cookie是再简单不过的了。
>
> 一个更微妙的层面上，然而，相信在cookies中存储的任意信息绝对不是一个好主意。 你永远不知道谁一直在作怪。
>
> 会话滞留 ：攻击者诱骗用户设置或者重设置该用户的会话ID。
>
> 例如，PHP允许在URL（如 `http://example.com/?PHPSESSID=fa90197ca25f6ab40bb1374c510d7a32` 等）中传递会话标识符。攻击者欺骗用户点击一个硬编码会话ID的链接，这回导致用户转到那个会话。
>
> 会话滞留已经运用在钓鱼攻击中，以诱骗用户在攻击者拥有的账号里输入其个人信息。 他可以稍后登陆账户并且检索数据。
>
> 会话中毒 ：攻击者通过用户提交设置会话数据的Web表单向该用户会话中注入潜在危险数据。
>
> 一个经典的例子就是一个站点在某个cookie中存储了简单的用户偏好（比如一个页面背景颜色）。 攻击者可以诱骗用户点击一个链接来提交背景颜色，实际上包含了一个XSS攻击。 如果颜色没有转义，那么就可以再把恶意代码注入到用户环境中。

解决方案

有许多基本准则能够保护您不受到这些攻击：

> 不要在URL中包含任何session信息。
>
> Django的session框架（参见 `第十四章 <../chapter14/>` __ ）根本不会容许session包含在URL中。
>
> 不要直接在cookie中保存数据。 相反，存储一个在后台映射到session数据存储的session ID。
>
> 如果使用Django内置的session框架（即 `request.session` ），它会自动进行处理。 这个session框架仅在cookie中存储一个session ID，所有的session数据将会被存储在数据库中。
>
> 如果需要在模板中显示session数据，要记得对其进行转义。 可参考之前的XSS部分，对所有用户提交的数据和浏览器提交的数据进行转义。 对于session信息，应该像用户提交的数据一样对其进行处理。
>
> 任何可能的地方都要防止攻击者进行session欺骗。
>
> 尽管去探测究竟是谁劫持了会话ID是几乎不可能的事儿，Django还是内置了保护措施来抵御暴力会话攻击。 会话ID被存在哈希表里（取代了序列数字），这样就阻止了暴力攻击，并且如果一个用户去尝试一个不存在的会话那么她总是会得到一个新的会话ID，这样就阻止了会话滞留。

请注意，以上没有一种准则和工具能够阻止中间人攻击。 这些类型的攻击是几乎不可能被探测的。 如果你的站点允许登陆用户去查看任意敏感数据的话，你应该 总是 通过HTTPS来提供网站服务。 此外，如果你的站点使用SSL，你应该将 `SESSION_COOKIE_SECURE` 设置为 `True` ，这样就能够使Django只通过HTTPS发送会话cookie。

邮件头部注入

邮件头部注入 ：SQL注入的兄弟，是一种通过劫持发送邮件的Web表单的攻击方式。 攻击者能够利用这种技术来通过你的邮件服务器发送垃圾邮件。 在这种攻击面前，任何方式的来自Web表单数据的邮件头部构筑都是非常脆弱的。

让我们看看在我们许多网站中发现的这种攻击的形式。 通常这种攻击会向硬编码邮件地址发送一个消息，因此，第一眼看上去并不显得像面对垃圾邮件那么脆弱。

但是，大多数表单都允许用户输入自己的邮件主题（同时还有from地址，邮件体，有时还有部分其他字段）。 这个主题字段被用来构建邮件消息的主题头部。

如果那个邮件头部在构建邮件信息时没有被转义，那么攻击者可以提交类似 `"hello\ncc:spamvictim@example.com"` （这里的 `"\n"` 是换行符）的东西。 这有可能使得所构建的邮件头部变成：

```
To: hardcoded@example.com
Subject: hello
cc: spamvictim@example.com
```

就像SQL注入那样，如果我们信任了用户提供的主题行，那样同样也会允许他构建一个头部恶意集，他也就能够利用联系人表单来发送垃圾邮件。

解决方案

我们能够采用与阻止SQL注入相同的方式来阻止这种攻击： 总是校验或者转义用户提交的内容。

Django内建邮件功能（在 `django.core.mail` 中）根本不允许在用来构建邮件头部的字段中存在换行符（表单，收件地址，还有主题）。 如果您试图使用 `django.core.mail.send_mail` 来处理包含换行符的主题时，Django将会抛出BadHeaderError异常。

如果你没有使用Django内建邮件功能来发送邮件，那么你需要确保包含在邮件头部的换行符能够引发错误或者被去掉。 你或许想仔细阅读 `django.core.mail` 中的 `SateMIMEText` 类来看看Django是如何做到这一点的。

目录遍历

目录遍历 ：是另外一种注入方式的攻击，在这种攻击中， 恶意用户诱骗文件系统代码对Web服务器不应该访问的文件进行读取和/或写入操作。

例子可以是这样的，某个视图试图在没有仔细对文件进行防毒处理的情况下从磁盘上读取文件：

```
def dump_file(request):
    filename = request.GET["filename"]
    filename = os.path.join(BASE_PATH, filename)
    content = open(filename).read()

    # ...
```

尽管一眼看上去，视图通过 `BASE_PATH` （通过使用 `os.path.join` ）限制了对于文件的访问，但如果攻击者使用了包含 `..` （两个句号，父目录的一种简写形式）的文件名，她就能够访问到 `BASE_PATH` 目录结构以上的文件。对她来说，发现究竟使用几个点号只是时间问题，比如这样： `../../../../../etc/passwd` 。

任何不做适当转义地读取文件操作，都可能导致这样的问题。 允许 写 操作的视图同样容易发生问题，而且结果往往更加可怕。

这个问题的另一种表现形式，出现在根据URL和其他的请求信息动态地加载模块。 一个众所周知的例子来自于Ruby on Rails。 在2006年上半年之前，Rails使用类似于 `http://example.com/person/poke/1` 这样的URL直接加载模块和调用函数。 结果是，精心构造的URL，可以自动地调用任意的代码，包括数据库的清空脚本。

解决方案

如果你的代码需要根据用户的输入来读写文件，你就需要确保，攻击者不能访问你所禁止访问的目录。

备注

不用多说，你 永远 不要在编写可以读取任何位置上的文件的代码！

Django内置的静态内容视图是做转义的一个好的示例（在 `django.views.static` 中）。这是相关代码：

```
import os
import posixpath

# ...

path = posixpath.normpath(urllib.unquote(path))
newpath = ''
for part in path.split('/'):
    if not part:
        # strip empty path components
        continue

    drive, part = os.path.splitdrive(part)
    head, part = os.path.split(part)
    if part in (os.curdir, os.pardir):
        # strip '.' and '..' in path
        continue

    newpath = os.path.join(newpath, part).replace('\\', '/')
```

Django不读取文件（除非你使用 `static.serve` 函数，但也受到了上面这段代码的保护），因此这种危险对于核心代码的影响就要小得多。

更进一步，URLconf抽象层的使用，意味着不经过你明确的指定，Django 决不会 装载代码。 通过创建一个URL来让Django装载没有在URLconf中出现的东西，是不可能发生的。

暴露错误消息

在开发过程中，通过浏览器检查错误和跟踪异常是非常有用的。 Django提供了漂亮且详细的debug信息，使得调试过程更加容易。

然而，一旦在站点上线以后，这些消息仍然被显示，它们就可能暴露你的代码或者是配置文件内容给攻击者。

还有，错误和调试消息对于最终用户而言是毫无用处的。 Django的理念是，站点的访问者永远不应该看到与应用相关的出错消息。 如果你的代码抛出了一个没有处理的异常，网站访问者不应该看到调试信息或者 任何 代码片段或者Python（面向开发者）出错消息。 访问者应该只看到友好的无法访问的页面。

当然，开发者需要在debug时看到调试信息。 因此，框架就要将这些出错消息显示给受信任的网站开发者，而要向公众隐藏。

解决方案

正如我们在第12章所提到的，Django的 `DEBUG` 设置控制这些错误信息的显示。 当你准备部署时请确认把这个设置为： `False` 。

在Apache和mod_python下开发的人员，还要保证在Apache的配置文件中关闭 `PythonDebug Off` 选项，这个会在Django被加载以前去除出错消息。

安全领域的总结

我们希望关于安全问题的讨论，不会太让你感到恐慌。 Web是一个处处布满陷阱的世界，但是只要有一些远见，你就能拥有安全的站点。

永远记住，Web安全是一个不断发展的领域。如果你正在阅读这本书的停止维护的那些版本，请阅读最新版本的这个部分来检查最新发现的漏洞。 事实上，每周或者每月花点时间挖掘Web应用安全，并且跟上最新的动态是一个很好的主意。 花费很少，但是对你网站和用户的保护确是无价的。

接下来？

你已经完成了我们安排的程序。 以下的附录内容中包含了可能在你的Djang项目中用得上的引用资源.

在运行你的Django网站时，无论是为你或几个朋友的小网站，或者是下一个google，我们祝你好运。